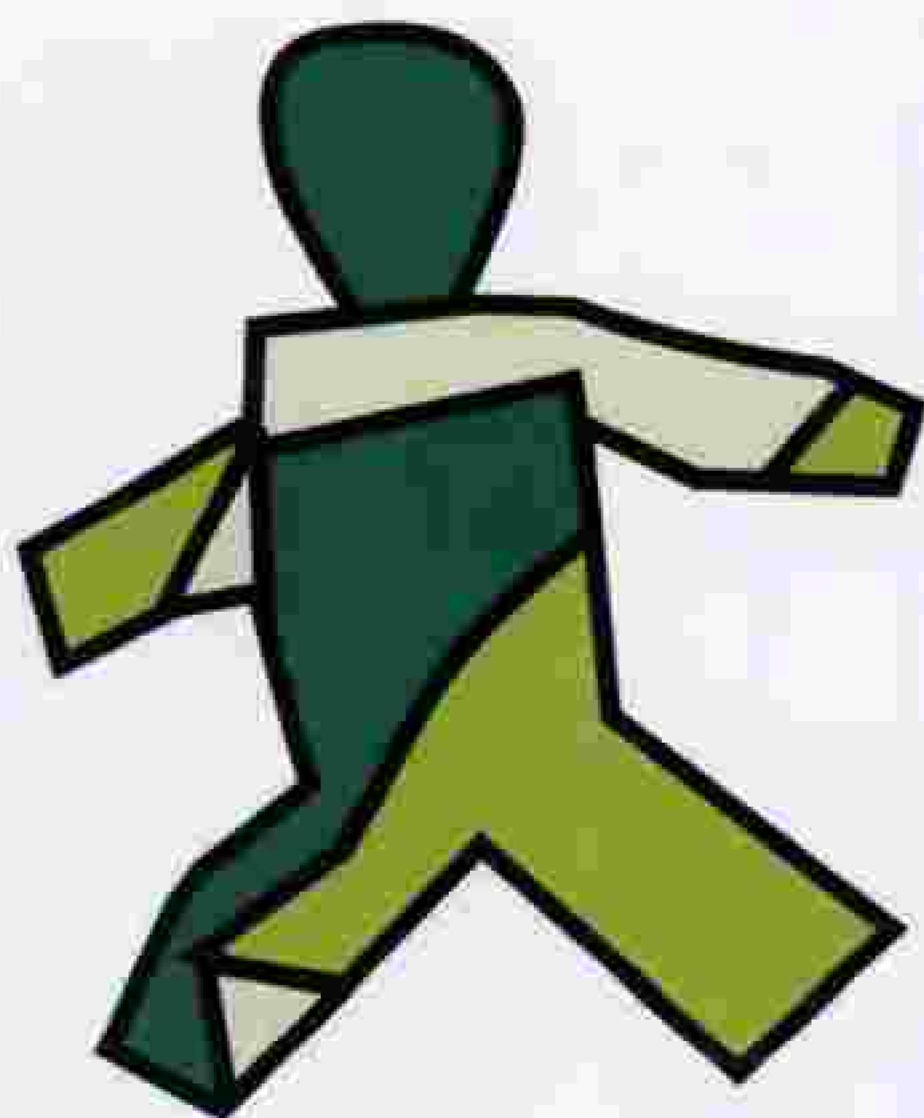
Manuale pratico di JAVA

la programmazione della piattaforma J2EE

SECONDA EDIZIONE

- networking avanzato e programmazione distribuita
- Java e database
- servlet e JSP
- programmazione a componenti EJB
- servizi avanzati di messaggistica



MokaByte

HOPS
tecniche nuove

Manuale pratico di Java: la programmazione della piattaforma J2EE
Paolo Aiello – Massimiliano Bigatti – Lavinio Cerquetti – Andrea Giovannini –
Gianluca Morello – Giovanni Puliti – Stefano Rossini – Nicola Venditti

http://www.mokabyte.it
e-mail: info@mokabyte.it

Redazione: tel. 0239090258 - 0239090257, fax 0239090255
e-mail: libri@tecnichenuove.com
Vendite: tel. 0239090251 - 0239090252, fax 0239090373,
e-mail: vendite-libri@tecnichenuove.com
http://www.tecnichenuove.com

ISBN 88-481-1582-9

Finito di stampare nel mese di novembre 2004
Stampa: New Press - Como

Hops Libri è un marchio registrato Tecniche Nuove

Indice generale

Prefazione ... vii

Capitolo 1 – RMI e la programmazione distribuita ... 1
Introduzione ... 1
Modelli distribuiti ... 2
La serializzazione ... 3
Architettura di RMI ... 7
RMI in pratica ... 9
RMI e il pooling degli oggetti remoti ... 14
RMI over IIOP ... 20
Riferimenti bibliografici ... 25

Capitolo 2 – Java e i database ... 27
Introduzione ... 27
Che cosa sono i database? ... 28
Accesso ai database: il punto di vista client ... 30
Introduzione a JDBC ... 31
Concetti di base di JDBC ... 37
JDBC 1.1 core API: struttura del package java.sql ... 52
JDBC 2.x core API ... 66
Estensioni di JDBC ... 83
Schema del database di demo utilizzato dal codice di esempio (in crea_mokadb.sql) ... 92

Capitolo 3 – Internazionalizzazione e localizzazione ... 95
Introduzione ... 95
La gestione delle font ... 98
Localizzazione: l'oggetto java.util.Locale ... 101
La formattazione ... 107
Formattazione di date e orari ... 112
Gestione dei messaggi ... 116
Confronto fra caratteri ... 118
La classe BreakIterator ... 122

Capitolo 4 – Java e XML ... 125
Introduzione ... 125
Fondamenti di XML ... 125
Elaborazione di documenti XML ... 127
SAX ... 128
DOM ... 132

JDOM 136
Data binding 142
Approfondimenti e benchmark 152
Validazione di documenti XML 165
XSL 171
Riferimenti bibliografici 177

Capitolo 5 – La Servlet API 179
La API 180
Implementazione e ciclo di vita di una servlet 182
Come interagire con una servlet: richieste e risposte 187
L'habitat di una servlet: il servlet context 195
Resource abstraction 200
Servlet e internazionalizzazione 202
Invocazione della servlet 203
Terminazione corretta delle servlet 208
Il mantenimento dello stato 211
Web Application 219
Gestione della sicurezza 222
La API 2.3: i filtri 233
Gli ascoltatori e ciclo di vita 240
Altre innovazioni nella API 2.3 244
Considerazioni sulla API 2.3 248
La Servlet API 2.4 248
Riferimenti bibliografici 256

Capitolo 6 – JSP e Web User Interface 257
Introduzione 257
Pagine web dinamiche 258
Un esempio di utilizzo 265
Le novità di JSP 2.0 e della Servlet API 2.4 308
Riferimenti bibliografici 311

Capitolo 7 – Java e CORBA 313
Introduzione 313
Object Management Group 314
I servizi CORBA 315
Le basi CORBA 316
Architettura CORBA 317
Invocazione CORBA 318
Interface Definition Language 322
Mapping IDL–Java 324
Un po' di pratica 332
CORBA Naming Service 340
Accesso concorrente a oggetti remoti 349
Utilizzo degli Object Adapter 355
Parametri per valore 372

CORBA runtime information 382
Callback 387
CORBA e i firewall 392
CORBA e J2EE 393

Capitolo 8 – Enterprise JavaBeans 397
La specifica EJB 397
Architettura 398
Tipi di Enterprise JavaBeans 399
Esempio applicativo 400
Strutturazione dei vari componenti 402
EJB dietro le quinte 407
Utilizzare gli EJB 408
I servizi di sistema 412
Gli entity bean 421
I session bean 435
La gestione delle transazioni 441
Gestione esplicita delle transazioni 450
La gestione delle eccezioni all'interno di un container EJB 455
EJB 2.0: CMP 2.0 e Abstract Persistent Model 459
Legami relazionali fra entity bean e relational field 464
EJB 2.0: la API Client Local 473
EJB 2.0: i Message Driven Bean 478
EJB 2.1: il Timer Service e Timer Bean 484
Riferimenti bibliografici 490

Capitolo 9 – Web service 491
Introduzione 491
Comunicare: SOAP e JAXM 500
Descrivere: WSDL e JAX-RPC 526
Catalogare e scoprire: UDDI e JAXR 542
Non solo SOAP/WSDL/UDDI 546
Riferimenti bibliografici 547

Capitolo 10 – Java Naming and Directory Interface API 549
Servizi di naming 549
Servizi di directory 550
LDAP 551
I dati in LDAP 552
JNDI 554
Il JNDI Service Provider (JNDI Driver) 555
Lavorare con JNDI 556
Leggere le proprietà di una entry 560
Ricerca di una entry 560
Modificare gli attributi di una entry 564
Memorizzare oggetti nel directory 565
Oggetti serializzabili 566

Oggetti Referenceable e Reference 567
Oggetti con attributi 569
Object Factory 570
Conclusione 571
Riferimenti bibliografici 572

Capitolo 11 – Java Connector Architecture 573
Introduzione 573
Integrazione e comunicazione 574
Gli elementi dell'architettura JCA 579
Common Client Interface (CCI) 581
Il deploy di un connettore 595
Conclusioni 597
Riferimenti bibliografici 598

Capitolo 12 – JMS 599
MOM (Message Oriented Middleware) 599
Modelli di messaging 601
Che cosa è JMS? 602
Una semplice applicazione PTP 612
Un'applicazione publish/subscribe: la chat JMS 614
Persistenza dei messaggi 619
I durable subscribers 622
La chat JMS "persistente" 624
L'applicazione MokaJmsSpy 625
Un'applicazione di compravendita biglietti 628
Riferimenti bibliografici 636

Capitolo 13 – JavaMail 637
Le piattaforme 638
Invio di un messaggio 639
Ricevere messaggi 642
Messaggi multipart 649
Ricezione di allegati 652
Conclusioni 656
Riferimenti bibliografici 657

Indice analitico 659

Prefazione

A distanza di circa un anno dall'uscita del volume dedicato al linguaggio, alle basi fondamentali della programmazione a oggetti e all'introduzione di alcuni concetti avanzati delle tecnologie Java, la collaborazione con l'editore Tecniche Nuove ci consente di presentare questo *Manuale pratico di Java: la programmazione della piattaforma J2EE*, che si inserisce nella serie di volumi dedicati al mondo Java che già da qualche anno, a cura di autori della redazione di MokaByte (http://www.mokabyte.it), sono stati pensati e realizzati con l'editore Hops.

I contenuti del libro presentano nuovi argomenti e riprendono in parte anche alcuni temi già trattati nella prima edizione del *Manuale*, che vengono qui sviluppati e aggiornati. Rispetto a quel libro, *Manuale pratico di Java: la programmazione della piattaforma J2EE* nasce in un contesto certamente diverso: più impegnativo, per la complessità ormai raggiunta da certe tecnologie ma, al contempo, più maturo perché i tre anni intercorsi dalla prima edizione hanno contribuito a definire in maniera più chiara lo scenario, hanno decretato l'affermazione di determinate tecnologie e l'obsolescenza di altre, hanno portato a un consolidamento e a una riorganizzazione della piattaforma J2EE.

Va detto, inoltre, come questi anni siano stati proficui per l'ulteriore approfondimento della professionalità degli autori che costituiscono la redazione di MokaByte: l'ingresso nel gruppo Imola (http://www.imolinfo.it), la collaborazione con il sito statunitense The Server Side (http://www.theserverside.com), i nuovi inserimenti, le ulteriori esperienze maturate nell'ambito dell'utilizzo della piattaforma J2EE in progetti "sul campo" per la gestione di sistemi complessi in ambito amministrativo e bancario, il feedback derivante dai commenti e dai riscontri sulle pubblicazioni fin qui realizzate, il continuo contatto con le migliaia di professionisti e appassionati che visitano il sito, animando le discussioni del forum e proponendo stimoli di approfondimento, hanno rappresentato utili spunti di miglioramento.

Il libro si rivolge a un pubblico di programmatori che abbia già una buona conoscenza delle basi della programmazione Object Oriented e di alcune tecnologie Java appartenenti alla piattaforma J2SE (a tal proposito, è inevitabile fare riferimento al *Manuale pratico di Java: dalla teoria alla programmazione*, uscito nel 2003 per lo stesso editore, come ideale complemento al presente volume). La piattaforma Java2 Enterprise Edition ha rappresentato uno sforzo di sintesi e di raccolta di tutte le tecnologie più avanzate disponibili per la realizzazione di applicazioni multilivello distribuite e ha raggiunto ormai affidabilità, coerenza e organizzazione.

L'ambito applicativo è quello della realizzazione di grandi applicazioni che funzionino in rete su più livelli e che possano soddisfare le esigenze di un gruppo organizzato, di un'azienda, di un'industria, di un'amministrazione pubblica, di un sistema bancario. Le numerose tecnologie lato server e lato client, e i diversi framework che ne consentono l'integrazione, fanno di J2EE uno strumento completo e flessibile, ma anche complesso. Era pertanto importante fornire, insieme alla necessaria teoria, anche una serie di esempi che da un lato illustrassero praticamente le tecniche in oggetto, dall'altro fornissero un punto di partenza per l'eventuale sviluppo di applicazioni o parti di applicazioni funzionanti.

La presentazione front-end dei contenuti, la memorizzazione e la persistenza back-end dei dati, l'integrazione con XML e con altri protocolli orientati al web per la realizzazione di servizi, il livello intermedio di elaborazione secondo la logica applicativa, la possibilità di gestire sistemi legacy e i molti altri aspetti delle applicazioni enterprise distribuite hanno oggi a disposizione numerose tecnologie sicure, stabili e scalabili, coerentemente inquadrate nella piattaforma Java2 Enterprise Edition. Nella trattazione, quindi, una certa importanza è sempre data agli aspetti architetturali: scegliere le tecnologie e i componenti nella combinazione e nell'integrazione più adatta al problema che l'applicazione deve risolvere e gestire.

Ci auguriamo che il volume riesca a trasmettere le metodologie e gli strumenti necessari alla realizzazione di applicazioni enterprise eleganti ed efficaci e che possa contribuire a una ulteriore diffusione delle tecnologie Java proprio nell'ambito in cui esse dimostrano sicurezza, scalabilità e stabilità.

la Redazione di MokaByte

Capitolo 1

RMI e la programmazione distribuita

GIOVANNI PULITI

Introduzione

Questo capitolo è dedicato alla programmazione distribuita tramite la Remote Method Interface API, il protocollo di comunicazione e invocazione introdotto con il JDK 1.1 e divenuto ormai una delle colonne portanti del cosiddetto *Java distributed computing*. Per programmazione distribuita si intende tipicamente quel particolare paradigma computazionale in cui una applicazione viene organizzata in moduli differenti localizzati in spazi di indirizzamento diversi fra loro.

In questo contesto la Remote Method Invocation (RMI) consente di realizzare applicazioni distribuite in cui un programma A è in grado di invocare i metodi di un oggetto B in esecuzione su un computer remoto: in tale scenario, per semplicità, si definisce *client* il programma chiamante, mentre il remoto è detto *server*.

Molti sono i vantaggi derivanti dall'adozione di tale modello: un sensibile miglioramento delle prestazioni complessive, una maggiore semplicità nella gestione delle risorse distribuite, e un sostanziale incremento delle potenzialità operative. Ad esempio si può pensare di suddividere un processo computazionalmente pesante in sottoroutine più piccole ed eseguire tali "pezzi di applicazione" su macchine diverse ottenendo una drastica diminuzione del tempo complessivo di esecuzione.

Nel caso in cui invece l'efficienza non sia l'obiettivo principale, si può comunque trarre vantaggio da una organizzazione distribuita, potendo gestire meglio e più semplicemente le varie risorse localizzate nei differenti spazi di indirizzamento. Si pensi per esempio a una strutturazione a tre livelli (3-Tier) per la gestione di database relazionali in Internet: dal punto di vista del client ci si deve preoccupare esclusivamente dell'interfacciamento con l'utente e dello scambio con il server remoto delle informazioni contenute nel database.

Un altro importante scenario in cui è utile l'utilizzo del modello distribuito è quello in cui si debbano realizzare applicazioni che si interfacciano con codice legacy: in tal caso si può pensare

di inglobare gli applicativi esistenti (*legacy* appunto) in oggetti remoti e pilotarne in tal modo le funzionalità da un client Java. In realtà, per questo genere di integrazioni si preferisce spesso utilizzare tecnologie come CORBA, dato che RMI richiede l'utilizzo esclusivo di Java come linguaggio di sviluppo, cosa che rende difficile l'integrazione con programmi scritti in altri linguaggi (alla fine del capitolo viene presentata l'evoluzione di RMI, basata sul protocollo IIOP, che risolve in parte questo problema di interoperabilità).

Modelli distribuiti

Uno dei requisiti fondamentali per implementare un sistema distribuito è disporre di un sistema di comunicazione fra macchine diverse, basato su standard e protocolli prestabiliti.

In Java la gestione dei socket è un compito relativamente semplice, tanto che si possono realizzare in maniera veloce sistemi di comunicazione con i quali scambiare informazioni in rete o controllare sistemi remoti. L'implementazione di una connessione via socket risolve però solo il problema di fondo (come instaurare la connessione) ma lascia in sospeso tutta la parte di definizione delle modalità di invocazione e dei vari protocolli per lo scambio delle informazioni.

Prima dell'introduzione di RMI, erano già disponibili strumenti per l'esecuzione di codice remoto, basti pensare alla *Remote Procedure Call* (RPC): con questa tecnologia è possibile gestire procedure facenti parte di applicazioni remote rispetto al chiamante. Le RPC hanno visto il massimo del loro successo nei sistemi Unix e sono strettamente legate al concetto di processo, ma male si inseriscono nel contesto del paradigma basato sugli oggetti. È questo il motivo principale alla base dell'esigenza di una tecnologia apposita, come RMI, per la gestione di oggetti distribuiti.

In realtà il panorama della progettazione e gestione di oggetti distribuiti offre valide alternative, come ad esempio DCOM (estensione di COM proprietaria di Microsoft) o CORBA: dato che una trattazione approfondita delle possibili alternative esula dagli scopi di questo capitolo, si può brevemente dire che la scelta può ricadere su RMI nel caso in cui si voglia implementare, in maniera semplice e veloce, una struttura a oggetti distribuiti *full-Java* (sia il lato client che quello server devono essere realizzati obbligatoriamente utilizzando tale linguaggio).

Interoperabilità e RMI

Purtroppo la semplicità di RMI non sempre può essere adottata per la realizzazione di applicazioni distribuite. Il protocollo di comunicazione è stato progettato volutamente per effettuare alcune operazione alquanto complesse, in modo da garantire quelli che poi sono divenuti i punti di forza di RMI, come gestione della sicurezza sulle invocazioni remote, garbage collector distribuito e così via.

La presenza di network eterogenei di cui non è possibile a priori conoscere la reale organizzazione (router che effettuano natting di sottoreti, firewall di vario tipo, per citare due casi), impedisce la comunicazione fra client e server RMI, tanto da rendere impossibile l'adozione di tale protocollo.

In realtà queste problematiche si presentano in vario modo con tutti i protocolli di computazione distribuita, dato che tutti si basano su sistemi, convenzioni, sintassi e semantiche di comunicazione più o meno proprietarie.

Questa grossa limitazione, unitamente alla sempre crescente esigenza di poter realizzare in modo semplice e affidabile sistemi interconnessi, ha dato vita a una fervente attività di ricerca ed evoluzione tecnologica, portando tra l'altro allo sviluppo dei web services (si veda il Capitolo 9).

Basati su standard aperti come HTTP e XML, i messaggi (invocazione e dati) in questo caso possono passare da ogni nodo della rete senza problemi di nessun tipo: i web services per questo motivo sono lo strumento principe da utilizzare quando si vuol realizzare interconnessione e interoperabilità fra sistemi eterogenei, distribuiti e remoti.

Con la nascita dei web services, in molti nel mondo Java si sono chiesti come questi potessero integrarsi nella piattaforma Java e quale fosse il loro reale utilizzo, dato che per certi versi potrebbero essere considerati come un'alternativa a modelli di programmazione distribuita già presenti, primo fra tutti EJB che, basato sul protocollo RMI, rappresenta al momento lo strumento più potente e importante del mondo J2EE.

Per poter dare una risposta completa sarebbe necessario probabilmente molto più spazio di quello qui a disposizione. Una risposta sintetica ma al tempo stesso sufficientemente esauriente e chiara è quella che dice di utilizzare i web services quando interoperabilità e connessione di sistemi eterogenei sono il principale obiettivo da conseguire, mentre RMI o EJB sono gli strumenti da adottare nel caso in cui non sia importante realizzare sistemi distribuiti basati su reti geografiche ma piuttosto siano importanti aspetti come transazionalità, sicurezza, pooling di risorse e così via, tutte caratteristiche su cui si basa fortemente EJB.

La serializzazione

Il meccanismo base utilizzato da RMI per la trasmissioni dei dati fra client e server è quello della serializzazione: è quindi sicuramente utile soffermarsi su questo importante sistema di trasmissione prima di affrontare nello specifico la Remote Method Invocation.

Grazie all'estrema semplicità con cui permette il flusso di dati complessi all'interno di uno stream, la serializzazione spesso viene utilizzata anche indipendentemente da applicazioni RMI, e quindi quanto verrà qui detto resta di utilità generale.

L'obiettivo principale della serializzazione è permettere la trasformazione in modo semplice di oggetti e strutture di oggetti in sequenze di byte manipolabili con i vari stream del package java.io.

Ad esempio, grazie alla serializzazione è possibile inviare strutture dati di complessità arbitraria tramite un socket (utilizzando gli stream associati al socket stesso), oppure salvarli su file al fine di mantenere la persistenza.

La scrittura su stream avviene mediante il metodo writeObject() appartenente alla classe ObjectOutputStream. Ad esempio, volendo salvare su file un'istanza di una ipotetica classe Record, si potrebbe scrivere

```
Record record = new Record();
FileOutputStream fos = new FileOutputStream("data.ser");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(record);
```

dove si è salvato su file binario (data.ser) un oggetto di tipo Record.

L'operazione, in questo caso, è stata fatta in due fasi: creazione di uno stream di serializzazione prima, e associazione di tale stream a un comune FileOutputStream.

In modo altrettanto semplice si può effettuare l'operazione opposta che permette la trasformazione da stream a oggetto

```
FileInputStream fis = new FileInputStream("data.ser");
ObjectInputStream ois = new ObjectInputStream(fis);
record = (Record)ois.readObject();
ois.close();
```

In questo caso si utilizza la classe ObjectInputStream e il metodo readObject(), il quale restituisce un oggetto di tipo Object, rendendo necessaria una operazione di conversione (cast esplicito).

In entrambi i casi le operazioni di lettura e scrittura devono essere inserite in appositi blocchi try–catch al fine di prevenire possibili problemi di lettura scrittura o di conversione.

Per poter serializzare un oggetto, un gruppo di oggetti, una struttura di complessità arbitraria, si utilizza sempre la medesima procedura e non ci sono particolari differenze di cui tener conto, a patto che l'oggetto sia serializzabile: per rispettare questo vincolo un oggetto deve implementare l'interfaccia Serializable.

Per questo, ad esempio, l'oggetto Record visto nell'esempio di cui sopra potrebbe essere così definito:

```
public class Record implements Serializable {
    private String firstName;
    private String lastName;
    private int phone;

    public Record (String firstName, String lastName, int phone) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.phone = phone;
    }
}
```

La regola della serializzazione è ricorsiva, per cui un oggetto, per essere serializzabile, deve contenere esclusivamente riferimenti a oggetti serializzabili.

La maggior parte delle classi contenute all'interno del JDK è serializzabile, fatta eccezione per alcuni casi particolari: non sono serializzabili tutte le classi che inglobano al loro interno strutture dati binarie dipendenti dalla piattaforma, come ad esempio molti degli oggetti dell'API JDBC. In questo caso infatti i vari oggetti contengono al loro interno puntatori a strutture dati o codice nativo utilizzato per la comunicazione con lo strato driver del database.

Per sapere se un dato oggetto sia serializzabile o meno si può utilizzare il tool Serial Version Inspector (comando serialver), messo a disposizione dal JDK, passando ad esso il nome completo della classe da analizzare.

Ad esempio, per verificare che la classe java.lang.String sia serializzabile si può scrivere da linea di comando la seguente istruzione

```
serialver java.lang.String
```

che restituisce il serialVersionUID dell'oggetto

```
java.lang.String: static final long serialVersionUID = -6849794470754667710L;
```

Invece tramite

```
serialver - show
```

si manda in esecuzione la versione con interfaccia grafica di tale strumento (fig. 1.1).

La serializzazione e la trasmissione degli oggetti

Benché il suo utilizzo sia relativamente semplice, la serializzazione nasconde alcuni aspetti importanti relativamente alla trasmissione degli oggetti.

Per quanto visto finora si potrebbe immaginare che la serializzazione permetta la trasmissione di oggetti per mezzo di stream: in realtà questa concezione è quanto mai errata, dato che a spostarsi sono solamente le informazioni che caratterizzano un'istanza di un particolare oggetto.

Ad esempio durante la trasmissione in un socket l'oggetto non viene mai spostato fisicamente ma ne viene inviata solo la sua rappresentazione e successivamente viene ricreata una copia identica dall'altra parte del socket: al momento della creazione di questa copia il runtime creerà un oggetto nuovo, riempiendo i suoi dati con quelli ricevuti dal client.

Risulta ovvio quindi che, al fine di consentire questo spostamento virtuale, su entrambi i lati, sia server che client, debba essere presente il codice relativo all'oggetto: il runtime quindi deve poter disporre dei file .class necessari per istanziare l'oggetto, o deve poterli reperire in qualche modo.

Il serialVersionUID della classe serve proprio per identificare di quale tipo di oggetto siano i dati prelevati dallo stream. Si tenga presente che, nella trasmissione delle informazioni relative all'oggetto sono inviati solamente quei dati realmente significativi della particolare istanza. Per questo non vengono inviati i metodi (che non cambiano mai), le costanti, le variabili con specificatore static che formalmente sono associate alla classe e non alla istanza, e quelle identificate con la parola chiave transient.

Figura 1.1 – *Il tool Serial Version Inspector è disponibile anche in versione grafica.*



Tale keyword qualifica una variabile non persistente, ovvero una variabile il cui valore non verrà inviato nello stream durante la serializzazione. Il valore assunto da una variabile di questo tipo dipende da come essa è stata definita. Ad esempio, supponendo di scrivere

```
transient Integer  Int = new Integer(10);
```

al momento della deserializzazione alla variabile Int verrà impostato il valore 10. Se invece si fosse scritto

```
transient Integer  Int;
```

durante la fase di deserializzazione Int assumerebbe il proprio valore di default, che per tutte le variabili di tipo reference è null, mentre per i tipi primitivi corrisponde al valore base (0 per gli int, false per i boolean, 0.0 per i float e così via).

L'interfaccia Externalizable

Riconsiderando l'esempio visto in precedenza, la classe Record viene serializzata e deserializzato su uno stream. In questo caso il processo di trasformazione da oggetto a sequenza di byte è effettuato utilizzando le procedure standard di conversione della JVM. Anche gli oggetti contenuti all'interno di Record sono trasformati in modo ricorsivo utilizzando le medesime tecniche.

La potenza di questa soluzione sta proprio nella semplicità con cui permette di effettuare tali operazioni di conversione; in alcuni casi però può essere utile ricorrere a procedure particolari di trasformazione. Ad esempio magari si desidera svolgere le operazioni di conversione su una delle variabili prima di effettuare il salvataggio su file di tutta la classe. In questi casi si può ricorrere all'interfaccia Externalizable e personalizzare in tal modo il processo di serializzazione.

Implementando tale interfaccia, e in particolare ridefinendo il corpo dei due metodi readExternal() e writeExternal(), si potranno definire fin nei minimi dettagli tutti gli aspetti della conversione dell'oggetto.

Ad esempio si potrebbe scrivere:

```
public class Record implements Externalizable {
    String Name;
    public MyObject(String n) {
        Name = n;
    }

    // salva i dati in modo personalizzato
    public void writeExternal(ObjectOutput out) {
        ...
    }

    // legge i dati in modo personalizzato
    public void readExternal(ObjectInput in) {
```

```
        ...
    }

}
```

A questo punto l'oggetto Record potrà essere serializzato e deserializzato in maniera standard, ma con una procedura di conversione particolare.

Architettura di RMI

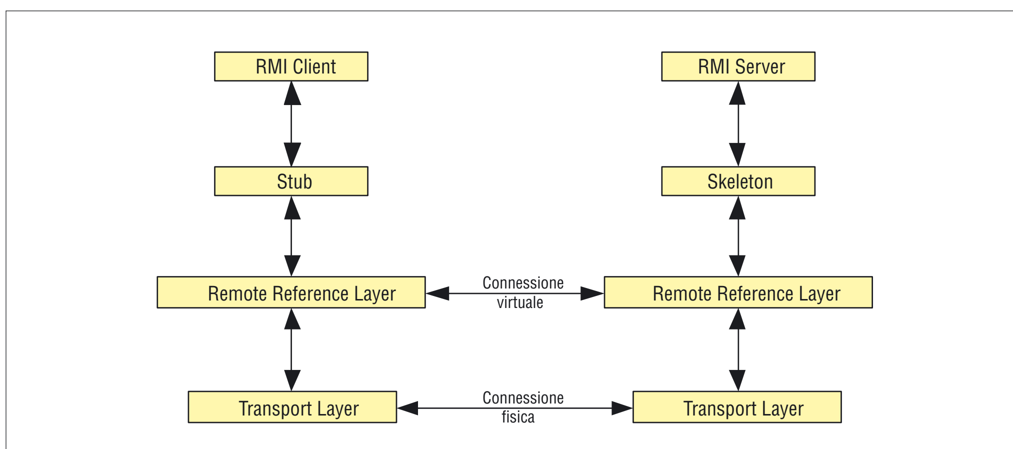
In fig. 1.2 è riportata la struttura tipica di una applicazione RMI: è possibile notare come essa sia organizzata orizzontalmente in strati sovrapposti, e in due moduli verticali paralleli fra loro.

Questa suddivisione verticale vede da una parte il lato client, e dall'altra il server: il primo è quello che contiene l'applicazione che richiede il servizio di un oggetto remoto, che a sua volta diviene il servente del servizio RMI.

Lo strato più alto del grafico è costituito su entrambi i lati (client e server) da una applicazione eseguita sulla Java Virtual Machine in esecuzione su quel lato: nel caso del client si tratta di una applicazione che effettua le chiamate ai metodi di oggetti remoti, i quali poi sono eseguiti dall'applicazione remota. Questa ha quindi un ciclo di vita indipendente dal client che di fatto ignora la sua presenza.

Subito sotto il livello "applicazione" troviamo i due elementi fondamentali dell'architettura RMI, ovvero lo stub e lo skeleton. Questi due oggetti forniscono una duplice rappresentazione dell'oggetto remoto: lo stub rappresenta una simulazione locale sul client dell'oggetto remoto che però, grazie allo skeleton, vive e viene eseguito sul lato server; i due elementi quindi non sono utilizzabili separatamente.

Figura 1.2 – *Organizzazione a strati della architettura lato client e lato server di RMI.*



Da un punto di vista funzionale, il client, dopo aver ottenuto un reference dell'oggetto remoto (lo stub di tale oggetto), ne esegue i metodi messi a disposizione per l'invocazione remota, in modo del tutto analogo al caso in cui l'oggetto sia locale. Si può quindi scrivere

```
OggettoRemoto.nomeMetodo();
```

Da un punto di vista sintattico non vi è quindi nessuna differenza fra un oggetto locale e uno remoto.

In conclusione il client (intendendo sia il programma che il programmatore della applicazione lato client) non ha che in minima parte la percezione di utilizzare un oggetto remoto.

Passaggio di parametri in RMI

Uno dei grossi vantaggi nell'utilizzo di RMI consiste nella semplicità con cui si possono passare parametri durante l'invocazione dei metodi remoti e riceverne indietro i risultati: senza nessuna differenza rispetto al caso locale si può scrivere

```
Ris  = OggettoRemoto.nomeMetodo(param_1, ..., param_n);
```

Riconsiderando lo schema riportato nella fig. 1.2, si ha che i vari parametri vengono serializzati dalla virtual machine del client, inviati sotto forma di stream al server, il quale li utilizzerà in forma deserializzata per utilizzarli all'interno del corpo del metodo invocato. Anche il durante percorso inverso, ovvero quello che restituisce un risultato al client, viene effettuata una serializzazione e quindi una deserializzazione dei dati.

Il procedimento, che da un punto di vista teorico risulta essere piuttosto complesso, è molto semplice da utilizzare. L'unico vincolo di cui si deve tener conto è che i parametri passati e il risultato ricevuto siano oggetti serializzabili.

Si tenga presente che, durante la trasmissione bidirezionale dei dati, viene sempre effettuata una copia (clonazione) dei reference, per cui non si ha la possibilità di lavorare su aree di memoria fisse, esigenza questa del resto non necessaria dato che si opera in uno scenario distribuito. In questo contesto, particolarmente importante è la presenza di un garbage collector apposito per RMI, il quale provvede in modo automatico e trasparente a eseguire le opportune operazioni di ripulitura delle aree di memoria non più utilizzate.

Gli strati RRL e TL

Il lato server e client sono collegati col sottostante Remote Reference Layer (RRL) che a sua volta si appoggia al Transport Layer (TL).

Il primo dei due ha il compito di instaurare un collegamento logico fra i due lati, di codificare le richieste del client, inviarle al server, decodificare le richieste e inoltrarle allo skeleton.

Ovviamente nel caso in cui quest'ultimo fornisca dei risultati per il particolare tipo di servizio richiesto, il meccanismo di restituzione di tali valori avviene in maniera del tutto simile ma in senso opposto.

Al livello RRL viene instaurato un collegamento virtuale fra i due lati client e server, mentre fisicamente la connessione avviene al livello sottostante, il Transport Layer. Tale collegamento è

di tipo sequenziale ed è per questo che si richiede la serializzazione dei parametri da passare ai metodi.

Il collegamento virtuale dello strato RRL si basa su un protocollo di comunicazione generico e indipendente dal particolare tipo di stub o skeleton utilizzati: questa genericità permette di mantenere la massima indipendenza dal livello stub/skeleton, tanto che è possibile sostituire il RRL con versioni successive più ottimizzate.

Il protocollo di conversione delle invocazioni dei metodi, l'impacchettamento dei riferimenti ai vari oggetti, e tutto quello che concerne la gestione a basso livello sono operazioni a carico sia dello strato RRL, sia e soprattutto dal TL, in cui si perde la concezione di oggetto remoto e/o locale e i dati vengono semplicemente visti come sequenze di byte da inviare o leggere verso certi indirizzi di memoria.

Quando il TL riceve una richiesta di connessione da parte del client, localizza il server RMI relativo all'oggetto remoto richiesto: successivamente viene eseguita una connessione per mezzo di un socket appositamente creato per il servizio. Una volta che la connessione è stabilita, il TL passa la connessione al lato client del RRL e aggiunge un riferimento dell'oggetto remoto nella tabella opportuna. Solo dopo questa operazione il client risulta effettivamente connesso al server e lo stub è utilizzabile dal client.

Il TL è responsabile del controllo dello stato delle varie connessioni: se passa un periodo di tempo significativo senza che venga effettuato nessun riferimento alla connessione remota, si assume che tale collegamento non sia più necessario, e quindi viene disattivato. Mediamente il periodo di timeout scatta dopo 10 minuti.

L'ultimo livello che però non viene incluso nella struttura RMI, è quello che riguarda la gestione della connessine al livello di socket e protocolli TCP/IP. Questo aspetto segue le specifiche standard di networking di Java e non offre particolari interessanti in ottica RMI.

RMI in pratica

Si può ora procedere ad analizzare quali siano i passi necessari per realizzare una applicazione RMI. Tutte le classi e i metodi che si analizzeranno e in generale tutte le API necessarie per lavorare con RMI, sono contenute nei package java.rmi e java.rmi.server.

Anche se dal punto di vista della programmazione a oggetti sarebbe più corretto parlare di classi, in questo caso si parlerà genericamente di oggetti remoti e locali intendendo sia il tipo che la variabile.

A tal proposito, in base alla definizione ufficiale, si definisce remoto un oggetto che implementi l'interfaccia Remote e i cui metodi possano essere eseguiti da una applicazione client non residente sulla stessa macchina virtuale.

Un'interfaccia remota invece rende disponibile il set di metodi utilizzabili per l'invocazione a distanza; ovviamente non è necessario definire nell'interfaccia quei metodi a solo uso interno della classe. Si immagini quindi di definire MyServer un oggetto per il momento non remoto.

```
public class MyServer {
    public void String concat(String a, String b) {
```

```
        return a + b;
    }
}
```

Il metodo concat() in questo caso esegue una concatenazione fra i due argomenti passati in input restituendo in uscita la stringa risultante.

A parte il vincolo della serializzabilità dei parametri, non ci sono limiti alla complessità delle operazioni eseguibili all'interno di metodi remoti.

Dopo aver definito questa semplice classe per trasformarla nella versione remota si deve per prima cosa definire la sua interfaccia remota.

```
public interface MyServerInterface extends Remote {
    public String concat(String a, String b) throws RemoteException;
}
```

Come si può osservare da queste poche righe di codice, per definire un'interfaccia remota è necessario estendere la java.rmi.Remote: questa è una interfaccia vuota e serve solo per verificare durante l'esecuzione che le operazioni di invocazione remota siano plausibili.

È obbligatoria la gestione dell'eccezione java.rmi.RemoteException: infatti, a causa della distribuzione in rete, oltre alla gestione di eventuali problemi derivanti dalla normale esecuzione del codice (bug o incongruenze di vario tipo), si deve adesso proteggere tutta l'applicazione da anomalie derivanti dall'utilizzo di risorse remote: ad esempio potrebbe venire a mancare improvvisamente la connessione fisica verso l'host dove è in esecuzione il server RMI.

Definita l'interfaccia remota si deve modificare leggermente la classe di partenza, in modo che implementi questa interfaccia:

```
public class MyServerImpl implements MyServerInterface
                               extends UnicastRemoteObject {

    public MyServerImpl() throws RemoteException {
        …
    }

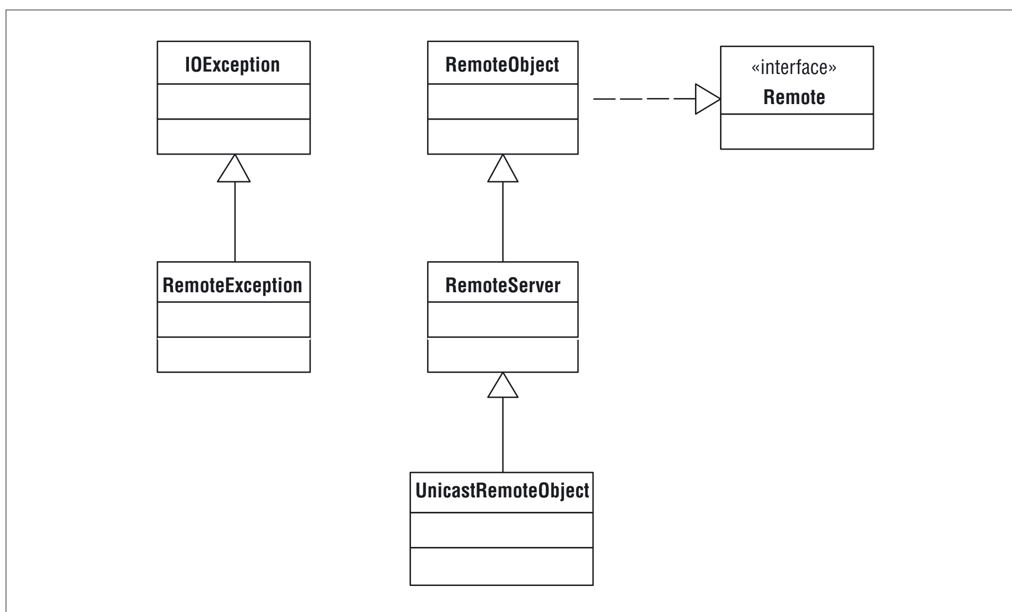
    public String concat(String a, String b) throws RemoteException{
        return a + b;
    }

}
```

Il nome della classe è stato cambiato a indicare l'implementazione dell'interfaccia remota; come si può notare, oltre a dichiarare di implementare l'interfaccia precedentemente definita, si deve anche estendere la classe UnicastRemoteObject.

Oltre a ciò, all'oggetto è stato aggiunto un costruttore di default il quale dichiara di propagare eccezioni RemoteException: tale passaggio non ha una motivazione apparente, ma è necessario per

Figura 1.3 – *Struttura gerarchica delle classi e delle interfacce più importanti in RMI.*



permettere al compilatore di creare correttamente tutte le parti che compongono il lato server (stub e skeleton).

La classe UnicastRemoteObject deriva dalle due classi, RemoteServer e RemoteObject: la prima è una superclasse comune per tutte le implementazioni di oggetti remoti (la parola Unicast ha importanti conseguenze come si avrà modo di vedere in seguito), mentre l'altra semplicemente ridefinisce hashcode() ed equals() in modo da permettere correttamente il confronto tra oggetti remoti.

L'uso della classe RemoteServer permette di utilizzare implementazioni di oggetti remoti diverse da UnicastRemoteObject, anche se per il momento quest'ultima è l'unica supportata.

L'organizzazione delle più importanti classi per RMI è raffigurata in fig. 1.3.

Dopo questa trasformazione, l'oggetto è visibile dall'esterno, ma ancora non utilizzabile secondo la logica RMI: si devono infatti creare i cosiddetti stub e skeleton.

Tali oggetti sono ottenibili in maniera molto semplice per mezzo del compilatore rmic, disponibile all'interno del JDK 1.1 e successivi: partendo dal bytecode ottenuto dopo la compilazione dell'oggetto remoto, questo tool produce stub e skeleton. Ad esempio, riconsiderando il caso della classe MyServerImpl, con una operazione del tipo

```
rmic MyServerImpl
```

si ottengono i due file MyServerImpl_stub.class e MyServerImpl_skel.class.

A questo punto si hanno a disposizione tutti i componenti per utilizzare l'oggetto remoto MyServerImpl: resta quindi da rendere possibile il collegamento tra client e server per l'invocazione remota.



Si definisce server RMI l'applicazione che istanzia un oggetto remoto e lo registra tramite una bind all'interno dell'RMIregistry. Il server RMI non è quindi l'oggetto che implementa la business logic, ma solamente una applicazione di servizio necessaria per attivare il meccanismo di invocazione remota.

Sul lato server, l'applicazione che gestisce lo skeleton deve notificare di possedere al suo interno un oggetto abilitato all'invocazione remota. Per far questo è necessario utilizzare il metodo statico java.rmi.Naming.bind() che associa all'istanza dell'oggetto remoto un nome logico con cui tale oggetto può essere identificato in rete.

Quindi, dopo aver creato una istanza dell'oggetto remoto tramite

```
MyServer server = new MyServer();
```

si provvede a effettuarne la registrazione utilizzando un nome simbolico

```
Naming.bind("MyServer", server);
```

Questa operazione, detta registrazione, può fallire e in tal caso viene generata una eccezione in funzione del tipo di errore. In particolare si otterrà una AlreadyBoundException nel caso in cui il nome logico sia già stato utilizzato per un'altra associazione, una MalformedURLException per errori nella sintassi dell'URL, mentre il runtime produrrà RemoteException per tutti gli altri tipi di errore legati alla gestione da remoto dell'oggetto.

Ogni associazione *nome logico – oggetto remoto* è memorizzata in un apposito registro detto RMIRegistry. In questo caso rmiregistry è anche il comando che lancia l'applicazione per gestire tale archivio, applicazione che deve essere lanciata sul lato server prima di ogni bind. Il client a questo punto è in grado di ottenere un reference all'oggetto con una ricerca presso l'host remoto utilizzando il nome logico con cui l'oggetto è stato registrato.

Ad esempio si potrebbe scrivere

```
MyServerInterface server;
String url = "//" + serverhost + "/MyServer";
server = (MyServerInterface) Naming.lookup(url);
```

e quindi utilizzare il reference per effettuare le invocazioni ai metodi remoti

```
System.out.println(server.concat("Hello ", "world!"));
```

Sul client per ottenere il reference si utilizza il metodo statico Naming.lookup(), che può essere considerato il corrispettivo alla operazione di bind sul server.

L'URL passato come parametro al lookup() identifica il nome della macchina che ospita l'oggetto remoto e il nome con cui l'oggetto è stato registrato.

Entrambe le operazioni di registrazione e di ricerca accettano come parametro un URL: il formato di tale stringa è la seguente:

```
rmi://host:port/name
```

dove host è il nome del server RMI, port la porta dove si mette in ascolto il registry, name il nome logico.

Sul server non è necessario specificare l'host, dato che per default assume l'indirizzo della macchina stessa sulla quale l'applicazione server RMI è mandata in esecuzione.

In entrambi i casi il numero della porta di default è la 1099 ma, se si specifica altrimenti, allora tale informazione dovrà essere passata al rmiregistry con il seguente comando:

```
rmiregistry numero_porta
```

Download del codice da remoto

Ogni volta che un parametro viene passato ad un metodo remoto, o viceversa ogni volta che si preleva un oggetto come risultato di una computazione remota, si dà vita a un processo di serializzazione o deserializzazione dell'oggetto in questione.

In realtà, come si è potuto vedere, l'oggetto serializzato non viene spostato dal client al server, ma vengono inviate nella rete solamente le informazioni necessarie per ricreare una copia dell'oggetto dal client al server (e viceversa).

Questo significa che sia il client che il server devono poter disporre dello stesso bytecode relativo all'oggetto serializzato in modo da poterne ricreare l'istanza. La soluzione più semplice è copiare fisicamente i vari file .class sia sul server che sul client: in questo caso si potrà essere sicuri che le varie operazioni di serializzazione e deserializzazione potranno essere effettuate correttamente.

Lo svantaggio di questa organizzazione risiede nel dovere ridistribuire tutti i file per ogni modifica delle varie classi. In alcuni casi questa soluzione è scomoda se non addirittura impraticabile.

RMI mette a disposizione un meccanismo molto potente che consente di scaricare dalla rete, tramite un file server HTTP, i file necessari per il funzionamento del client.

Nel caso delle applet, il classloader effettua una chiamata al motore HTTP del browser per scaricare tali file: da questo punto di vista non vi è differenza fra oggetti remoti che devono essere presenti in locale per la deserializzazione, e le normali classi Java necessarie per far funzionare l'applet.

Le classi sono localizzate automaticamente in Internet, facendo riferimento al codebase di provenienza (il quale tra l'altro è l'unico indirizzo verso il quale l'applet può connettersi).

Per quanto riguarda invece le applicazioni, la mancanza del browser complica le cose, dato che devono essere risolti due problemi: il primo è come effettuare il download vero e proprio delle classi, e il secondo come localizzare il server dal quale scaricare tale codice.

Per il primo aspetto esiste un oggetto apposito che effettua tale operazione: si tratta della classe RMIClassLoader parte integrante del package rmi.server e che può essere vista come una evoluzione della ClassLoader.

Per specificare l'indirizzo dell'host remoto dove è in funzione tale server si può procedere in due modi: o lo si inserisce direttamente in maniera stabile dentro il codice (*hardcoded URL*), oppure lo si passa al client come parametro di configurazione dall'esterno. Per rendere le cose ancora più semplici e automatiche si può fare in modo che sia il server, utilizzando il protocollo RMI, a comunicare al client l'indirizzo dove prelevare tali classi (nel caso delle applicazioni, non essendoci particolari vincoli, tale indirizzo potrà essere differente da quello server RMI). Per fare questo è necessario mandare in esecuzione il server RMI specificando nella opzione java.rmi.server.codebase l'URL presso cui prelevare via HTTP le classi necessarie.

La sintassi di esecuzione del server è la seguente

```
java –Djava.rmi.server.codebase = http://nome_host:port/rmi_dir/ ServerRmi
```

dove ServerRmi indica il nome della applicazione server RMI, mentre nome_host:port specifica l'indirizzo Internet e la porta dove è in funzione il server HTTP.

Tale demone dovrà avere accesso alle classi remote che dovranno essere posizionate nella directory rmi_dir/.

Sicurezza e download del codice

Nell'ambito della programmazione di rete, nasce il problema di garantire un sufficiente livello di sicurezza tutte le volte che si esegue del codice scaricato da remoto: chi garantisce che tale codice non esegua operazioni pericolose?

Java effettua in tal senso un controllo molto scrupoloso, grazie alla classe SecurityManager che, nel caso di RMI si chiama RMISecurityManager. Dal punto di vista del client, se nessun RMISecurityManager è stato installato, allora potranno essere caricate classi solamente dal classpath locale.

RMI e il pooling degli oggetti remoti

Se si ripensa per un momento alla modalità di pubblicazione di un oggetto remoto da parte del server RMI, si potrà osservare come la funzione di creazione e registrazione sia un compito totalmente a carico del server. Per una precisa scelta progettuale quindi, visto che la registrazione dell'oggetto avviene una volta sola, l'istanza dell'oggetto remoto sarà l'unica disponibile e quindi condivisa fra tutti i client possibili.

Pensata per semplificare al massimo il lavoro del programmatore, in molti casi però questa soluzione risulta essere troppo rigida e non sufficiente per supportare architetture distribuite complesse.

Una soluzione semplicistica potrebbe essere quella di istanziare un numero prefissato di oggetti per poter servire più client; ovviamente tale soluzione, oltre ad essere poco flessibile e per niente elegante, non risolve il problema della creazione on demand di oggetti remoti da parte del client.

Questo tipo di problema, che ricade nella sfera del pooling degli oggetti, può essere risolto in RMI tramite due tecniche: una basata sull'utilizzo di particolari pattern progettuali (soluzione

quindi non strettamente legata a RMI, ma di valenza generale) e una basata sull'utilizzo di una particolare interfaccia remota messa appositamente a disposizione per risolvere questo problema.



Il framework EJB che per certi versi può essere considerato come l'evoluzione di RMI nasce per risolvere una serie di problemi presenti nei modelli di programmazione distribuita come RMI, ma anche CORBA e DCOM. Infatti in EJB il problema del pooling degli oggetti viene risolto in modo molto potente ed elegante, demandando al container la gestione del numero degli oggetti remoti in esecuzione in base alle esigenze di sistema e alle richieste dei vari client.

RMI e il pattern Factory

Il problema del pooling di oggetti può essere risolto utilizzando una tecnica introdotta in RMI ad hoc, ma risulta essere alquanto complessa (vedi oltre): la programmazione per pattern ([GOF] [JPATTERN] [MBPATTERN]), permette di dar vita a interessanti soluzioni. Utilizzando il pattern Factory Method si può ottenere lo stesso risultato in modo molto più semplice ed elegante. Si supponga ad esempio di avere un'insieme base di oggetti strutturati come riportato in fig. 1.4.

Si tratta di una semplice gerarchia di classi per la gestione dei messaggi di log all'interno di una applicazione: per semplicità verrà preso in considerazione il caso in cui sia presente il solo metodo `log(String messaggio)` per la produzione di tali messaggi.

La classe base (`Logger`) serve pertanto a gestire messaggi di tipo generico, mentre le due classi derivate potrebbero implementare tecniche particolari per la gestione di messaggi verso file (classe `FileLogger`) e verso uno stream generico (classe `StreamLogger`).

Figura 1.4 – *Struttura gerarchica di base per la gestione di messaggi di log.*

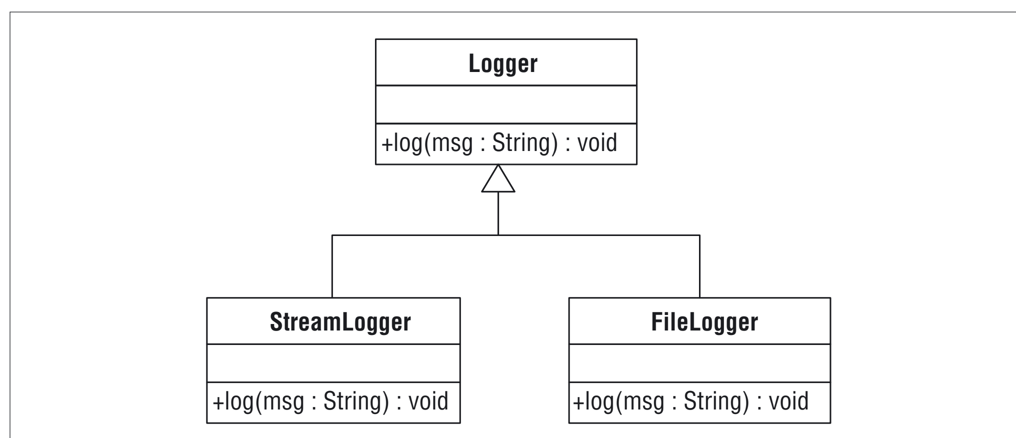
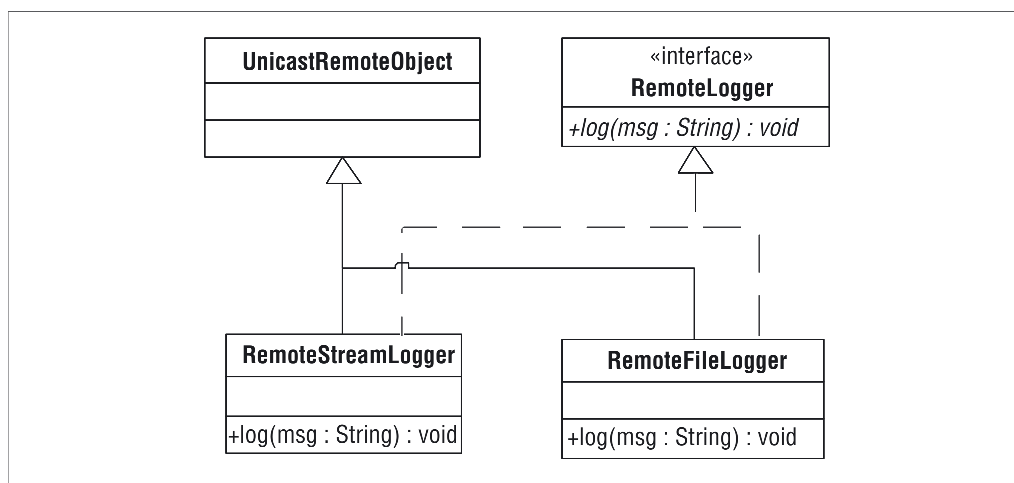


Figura 1.5 – *Versione remota delle classi per la gestione dei messaggi da client RMI.*



A tal punto si può immaginare che un ipotetico client possa aver bisogno di utilizzare i servizi di un oggetto di tipo FileLogger oppure StreamLogger, ad esempio per memorizzare alcune informazioni relative al suo funzionamento.

In uno scenario reale si potrebbe ipotizzare che ogni client debba o voglia poter produrre messaggi propri indipendentemente dagli altri, e che tali messaggi siano gestiti da un server centrale. Utilizzando RMI allora si dovranno per prima cosa creare *n* oggetti remoti, in teoria uno per ogni client che desideri fare log da remoto, e successivamente registrarli.

Per rendere remoti gli oggetti visti in precedenza è necessario modificare leggermente la gerarchia di classi, in direzione di un'organizzazione delle classi come quella riportata in fig. 1.5.

Come si può notare, la classe base è stata sostituita dall'interfaccia remota RemoteLogger, la quale, oltre a sopperire alla funzione di interfaccia standard, permette alle sottoclassi di essere oggetti remoti a tutti gli effetti.

Si è predisposta in tal modo la base per permettere l'utilizzo da client RMI di oggetti remoti. A questo punto si deve predisporre un meccanismo che permetta la creazione di tali oggetti anche da parte di client RMI.

La classe LoggerFactory che implementa il pattern Factory, è a tutti gli effetti un oggetto remoto come quelli visti in precedenza: tramite la sua registrazione, ogni client potrà ottenerne lo stub e invocarne il metodo remoto getLogger(), la cui implementazione è riportata di seguito.

```
public RemoteLogger getLogger(int type) throws RemoteException {
    RemoteLogger ret = null;

    if (type == 1)
        ret = new RemoteFileLogger();
```

```
        if (type == 2)
            ret = new RemoteStreamLogger();

        return ret;

    }
```

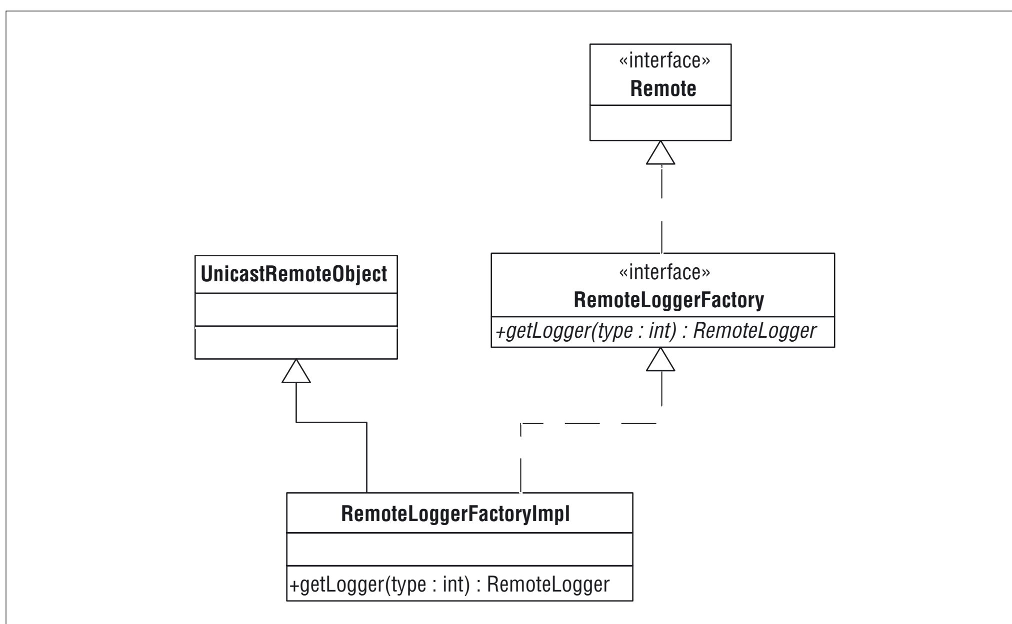
Tale metodo, controllando il valore del parametro passato, è in grado di restituire un oggetto — remoto — di tipo FileLogger e StreamLogger; più precisamente viene restituita un'interfaccia remota Logger, cosa che permette di mantenere una grande genericità.

Da un punto di vista organizzativo le classi remote che implementano il Factory sono strutturate secondo lo schema UML riportato in fig. 1.6. Il client invocando il metodo remoto getLogger() riceverà lo stub dell'oggetto remoto, il quale verrà eseguito sul server remoto, in perfetto accordo con il modello teorico di RMI.

Attivazione di oggetti remoti

L'utilizzo del pattern Factory risolve in maniera piuttosto agile alcune delle limitazioni imposte dal modello RMI: con tale tecnica infatti è possibile attivare oggetti al momento della effettiva necessità su richiesta del client ed in modo esclusivo.

Figura 1.6 – *Il factory remoto.*



Rispetto alla situazione standard, il problema dello spreco delle risorse è quindi ridotto, dato che, tranne che per il factory remoto, gli altri oggetti remoti sono inizializzati al momento dell'effettivo bisogno.

Con il rilascio della piattaforma Java 2 è stata introdotta una tecnica alternativa, che si basa su un altro tipo di oggetti remoti: invece che derivare da UnicastRemoteObject, in questo caso l'oggetto attivabile su richiesta del cliente estende direttamente la classe java.rmi.activation.Activatable.

La maggiore flessibilità che ne deriva permette un più razionale utilizzo delle risorse, senza sconvolgere di fatto l'organizzazione delle applicazioni, dato che dal punto di vista del client RMI il meccanismo di invocazione remota è identico al caso standard.

Le operazioni da fare per modificare un oggetto remoto riguardano, come accennato in precedenza, esclusivamente il lato server. L'oggetto remoto in questo caso deve essere modificato in modo che per prima cosa estenda la classe Activatable invece che la UnicastRemoteObject.

```
import java.rmi.*;
import java.rmi.activation.*;
public class Active extends Activatable
```

Si noti l'import del package java.rmi.activation al posto del java.rmi.server. Successivamente si deve procedere a rimuovere o commentare il costruttore vuoto che prima era obbligatoriamente necessario definire; al suo posto si dovrà mettere la versione riportata di seguito:

```
public Active(ActivationId id, MarshallObject data) throws Exception {
    super(id, data);
}
```

A questo punto formalmente l'oggetto non è più un oggetto remoto ma viene detto attivabile.

Per quanto riguarda l'applicazione server, vi è un'altra importante differenza: nel caso degli oggetti remoti essa doveva restare attiva per tutto il tempo necessario all'utilizzo degli oggetti remoti da parte dei client. Adesso invece gli oggetti remoti sono gestiti da un demone apposito (rmid), mentre il server deve solamente effettuare l'operazione di registrazione: dopo che tale installazione è conclusa il server può terminare la sua esecuzione. Per questo motivo in genere si utilizzano nomi del tipo setupXXX invece di serverXXX.

Il processo di installazione si traduce nel creare un ambiente di lavoro per l'oggetto attivabile (quello che si definisce ActivationGroup), impostando eventualmente delle variabili che verranno utilizzate al momento della inizializzazione dell'oggetto attivabile; successivamente tale setup-class provvede a registrare tale oggetto nel registro RMI.

Di seguito sono riportate sinteticamente le operazioni da svolgere per installare un oggetto attivabile.

```
import java.rmi.*;
import java.rmi.activation.*;
import java.util.Properties;
```

```
//installare un security manager appropriato

//creare un'istanza di ActivationGroup

Properties props;
props  = (Properties)System.getProperties().clone();
ActivationGroupDesc Agd;
Agd = new ActivationGroupDesc(props, null);
ActivationGroupID Agi;
Agi  = ActivationGroup.getSystem().registerGroup(Agd);
ActivationGroup.createGroup(Agi, Agd, 0);
```

Si deve infine creare una istanza di ActivationDesc la quale fornisce tutte le informazioni di cui il demone rmid necessita per creare le istanze vere e proprie degli oggetti attivabili.

Queste informazioni consistono di varie parti, come ad esempio il nome della classe, la collocazione del codice remoto e una istanza della classe che serve per passare una configurazione di inizializzazione all'oggetto remoto.

Ad esempio si può scrivere

```
ActivationDesc Desc;
Desc  = new ActivationDesc("EchoApp.EchoImpl", location, data);
```

La classe MarshalledObject, introdotta con il JDK 1.2, contiene un bytestream dove viene memorizzata un rappresentazione serializzata dell'oggetto passato al suo costruttore.

```
public MarshalledObject(Object obj) throws IOException
```

Il metodo get restituisce una copia non serializzata dell'oggetto originale. La modalità con cui vengono effettuate le operazioni di serializzazione e deserializzazione sono le stesse utilizzate durante il processo di marshalling dei parametri durante una invocazione RMI di un metodo remoto.

Durante la serializzazione, l'oggetto viene memorizzato con il codebase dell'URL da dove la classe eventualmente può essere scaricata. Inoltre, ogni oggetto remoto memorizzato all'interno del MarshalledObject è rappresentato con una istanza serializzata dello stub dell'oggetto medesimo.

La classe MarshalledObject è utile in tutti quei casi in cui si debbano implementare manualmente alcuni passaggi tipici di RMI, come il trasferimento di uno stub da client a server, oppure la serializzazione dello stesso secondo la semantica di serializzazione utilizzata in RMI.

Molto utile la possibilità di scaricare, per mezzo del metodo get, la classe corrispondente all'oggetto remoto se questa non è presente in locale al momento della lookup da parte del client RMI. Questa funzione infatti permette di risolvere uno dei problemi principali di RMI, ovvero la necessità di dover installare sul client il codice (.class) corrispondente allo stub dell'oggetto remoto, prima di effettuare una lookup.

RMI over IIOP

Come si è avuto modo di accennare in precedenza, RMI, rispetto all'analogo CORBA, ha il vantaggio principale della semplicità, ma vincola a dover utilizzare strato client e server in Java e a dover utilizzare sempre il medesimo protocollo di comunicazione JRMP, rispettando al contempo le modalità di interrogazione dei metodi e di marshalling dei parametri. Per questo motivo RMI offre semplicità ma non è in grado di mantenere interoperabilità con piattaforme differenti ovvero con applicazioni distribuite scritte in linguaggi diversi.

L'altro soggetto principale coinvolto in questo scenario è CORBA, il quale ha una connotazione completamente opposta rispetto a RMI: maggiore interoperabilità pagata al prezzo di una maggiore complessità. Con CORBA, una applicazione client Java può interagire con una serie di oggetti remoti scritti in altri linguaggi (C++, Delphi o altri per i quali sia disponibile un mapping IDL) e viceversa.

Dato che Java è diventato il dominatore incontrastato in ambito enterprise soprattutto nella realizzazione di applicazioni distribuite, portabili, ed integrate, si è fatta pressante la necessità di una qualche forma di integrazione fra le due tecnologie.

Le eventuali modifiche da apportare a RMI dovevano essere il più trasparenti possibili, sia per garantire la retrocompatibilità con il "vecchio RMI", sia perché un aumento del livello di complessità in RMI avrebbe significato limitare la sua caratteristica più importante, ovvero semplicità.

Fortunatamente le modifiche sono state effettuate in gran parte all'interno di CORBA, grazie all'introduzione di alcuni nuovi paradigmi (gli object by value e l'IDL-to-Java) garantendo al contempo sia la retrocompatibilità che l'integrazione con RMI.

L'unione dei due mondi

L'integrazione del mondo Java RMI e di CORBA è resa possibile grazie alla introduzione di RMI-IIOP dove IIOP indica Internet Inter Orb Protocol. Si tratta di un protocollo pensato in modo da garantire la comunicazione sullo strato TCP (Internet) e di connettori ORB differenti fra loro (IOP).

Sviluppato congiuntamente da IBM e SUN nel 1999, esso rappresenta il matrimonio fra RMI e CORBA. Lo scenario che si presentava prima della introduzione di RMI-IIOP era quello raffigurato in fig. 1.7: applicazioni distribuite sviluppate in Java RMI e basate sul protocollo JRMP (Java Remote Protocol) non erano in grado di interagire con applicazioni scritte con tecnologie differenti e viceversa.

L'introduzione di RMI-IIOP ha modificato le cose: se si osserva la fig. 1.8, si potrà notare come tale tecnologia consenta la comunicazione fra applicazioni Java RMI e RMI-IIOP e fra RMI-IIOP e CORBA. Di fatto non è ancora possibile connettere direttamente una applicazione Java RMI con una CORBA, ma questo nella maggior parte dei casi non è un problema, come si potrà vedere in seguito.

Anche il modo in cui tutto ciò sia possibile può apparire piuttosto misterioso, in realtà si tratta di una soluzione molto semplice. Il protocollo RMI-IIOP infatti è in grado di supportare in modo trasparente entrambi JRMP e IIOP. Un server o un client scritti per lavorare con RMI-IIOP non necessitano di essere riscritti o ricompilati per supportare l'uno o l'altro; per il client infatti è

Figura 1.7 – *Lo scenario delle applicazioni distribuite prima di RMI-IIOP.*

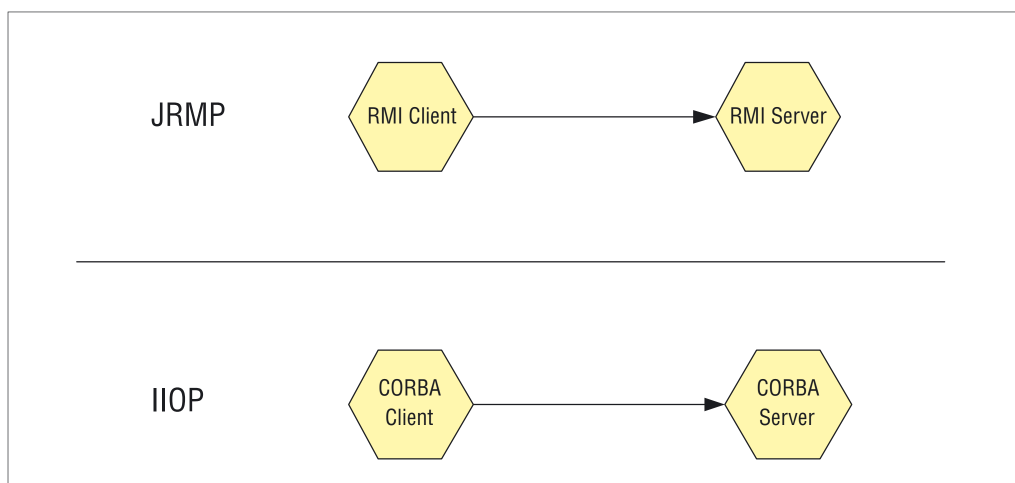
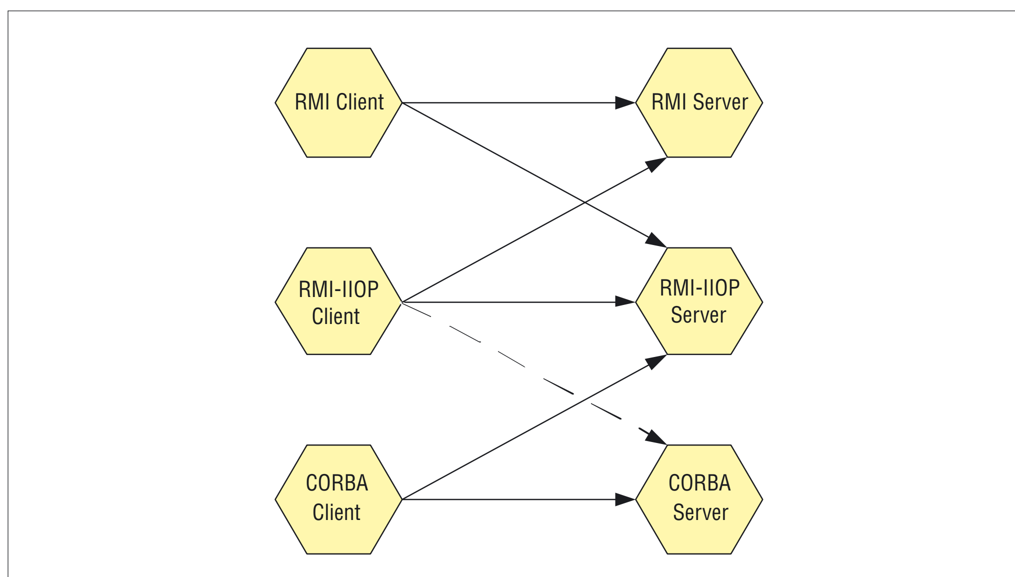


Figura 1.8 – *RMI-IIOP unisce i due strati applicativi: ogni linea diretta nella figura rappresenta la possibilità di un client di invocare un oggetto remoto in esecuzione sul server.*



sufficiente modificare alcuni parametri operativi della JVM o direttamente all'interno del codice Java, mentre un oggetto remoto (server-side) potrà essere esposto in modo da supportare entrambi i protocolli grazie al cosiddetto *dual-export* (vedi oltre).

La linea tratteggiata in fig. 1.8 rappresenta una situazione particolare: in determinate situazione può accadere che una applicazione client RMI-IIOP non sia in grado di invocare oggetti CORBA. Questo può avvenire fondamentalmente per la differente semantica dei tipi definiti dall'IDL CORBA che rappresenta un superset di quelli RMI-IIOP.

In alcuni casi, tipicamente in tutti quelli in cui un client Java RMI-IIOP debba integrarsi con oggetti CORBA preesistenti (legacy objects), può essere che non sia possibile la comunicazione.

Questa limitazione è spesso eliminabile grazie al compilatore *rmic*, presente nel JDK: tale strumento permette di creare il file IDL dagli oggetti Java RMI-IIOP. A partire dall'IDL si potrà generare un oggetto CORBA ad esempio in C++, oggetto che sarà invocabile da un client C++ CORBA, ma anche, e senza nessuna limitazione, da uno Java basato su RMI-IIOP.

Si potrebbe quindi dire che l'incompatibilità sia un problema esclusivamente di implementazione tale da non inficiare la comunicazione fra i due strati della fig. 1.8.

In quei casi in cui non sia possibile riscrivere l'oggetto CORBA e il mapping non permetta a RMI-IIOP di interagire con tale oggetto, come valida alternativa si può pensare di abbandonare completamente RMI e passare a Java IDL (vedi [JIDL]). Tramite Java IDL, una delle tecnologie introdotte con Java2, è possibile accedere a un oggetto CORBA direttamente da Java, senza passare per altri protocolli bridge intermediari. È anche possibile effettuare il contrario, ovvero esporre un oggetto Java al mondo CORBA, ma per tale scenario è sicuramente preferibile utilizzare RMI-IIOP.

Il punto di forza di RMI-IIOP è dato dal fatto che lascia inalterata la semantica di RMI (e anche la sintassi, eccezion fatta per alcune piccole modifiche del codice), aggiungendo alcune importanti modifiche a CORBA in modo da aumentarne le funzionalità: in CORBA infatti è stata introdotta con la versione 2.3 della specifica il passaggio dei parametri per valore (object by value) già presente in RMI tramite il concetto di serializzazione dei parametri e il Java-to-IDL mapping che consente il mapping dei tipi Java sui tipi CORBA.

Figura 1.9 – *Tramite Java IDL si può connettere il mondo Java direttamente con CORBA.*

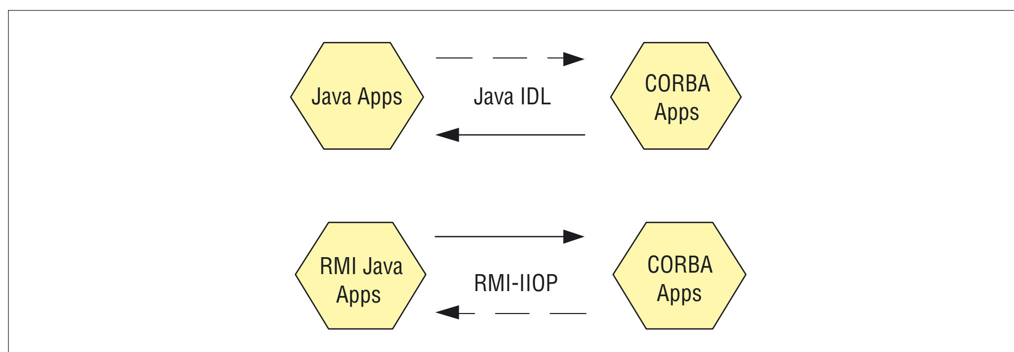
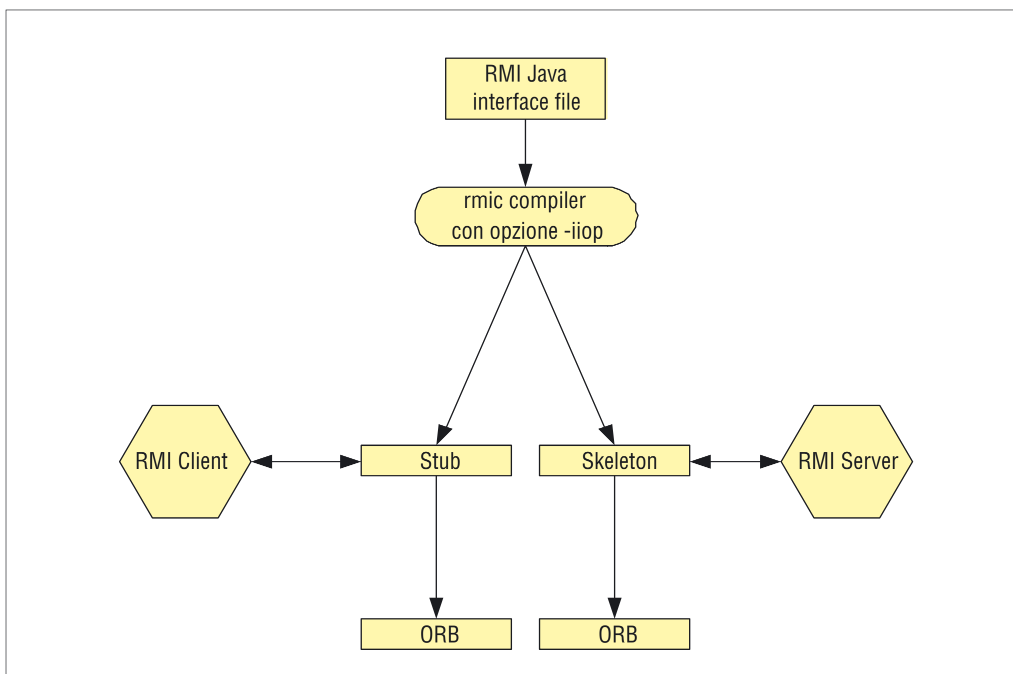


Figura 1.10 – *Il processo di sviluppo di un oggetto Java RMI-IIOP.*



Queste innovazioni, specie la prima, non devono essere sottovalutate, dato che sono di fatto variazioni molto profonde, accettate da OMG non senza qualche timore.

Creare un oggetto remoto

La realizzazione di un oggetto remoto conforme con lo standard RMI-IIOP è molto simile al caso basato sul protocollo JRMP. A parte alcune piccole differenze nel codice, che saranno affrontate successivamente, il processo di compilazione prevede l'utilizzo del parametro `–iiop` da passare al compilatore `rmic`. In tal modo il compilatore produce stub e skeleton conformi a IIOP invece che solamente a JRMP.

Ovviamente il passaggio a RMI-IIOP comporta alcune importanti differenze funzionali dovute alla presenza di CORBA: ad esempio in questo caso non è disponibile il garbage collector distribuito come in RMI. A parte questo, il programmatore non dovrebbe avere altra considerazione della differente architettura sottostante.

Questo è il punto in cui la teoria Java RMI finisce e ci si addentra nel mondo di CORBA, e quindi si rimanda alla bibliografia per maggiori approfondimenti in merito.

Da un punto di vista implementativo per realizzare un oggetto remoto la prima cosa da fare è estendere dalla classe `PortableRemoteObject` al posto della `Remote` di RMI.

Si supponga ad esempio di voler creare un oggetto remoto che restituisce l'ora locale del server in cui viene eseguito. Nella versione RMI questo oggetto potrebbe essere così definito

```
public class MyRemoteClock  implements MyRemoteClockRemote {
    public String getRemoteDate() throws RemoteException{
        return new Date().toString();
    }
}
```

ma in RMI-IIOP dovrebbe essere trasformato come segue

```
public class MyRemoteClock  extends PortableRemoteObject implements  MyRemoteClock {
    public String getRemoteDate() throws RemoteException{
        return new Date().toString();
    }
}
```

L'altra modifica lato server da effettuare riguarda la procedura di registrazione dell'oggetto nel registry. Dato che in questo caso infatti viene utilizzato JNDI (vedi Capitolo 10) come sistema di name service, al posto della istruzione

```
naming.rebind(nome-oggetto, istanza-oggetto)
```

si dovrà utilizzare

```
naming.rebind(nome-oggetto, istanza-oggetto)
```

Infine dopo aver generato stub e skeleton con il comando

```
rmic –iiop MyRemoteClock
```

si dovrà procedere ad attivare il registry, utilizzando il comando `tnameserv` (al posto di `rmiregistry`).

Su lato client invece si dovranno effettuare solo le modifiche relative al differente modo di effettuare la lookup dell'oggetto remoto, basato su JNDI invece che sul RMIregistry. In questo caso si dovrà scrivere

```
InitialContext ic = new InitialContext(env)
Object obj = ic.lookup(nome-oggetto);
MyRemoteClock mrc = (MyRemoteClock) PortableRemoteObject.narrow(obj, MyRemoteClock.class);
```

A questo punto, per il client sarà possibile utilizzare l'oggetto `mrc` in modo del tutto analogo al caso RMI, ignorando se l'implementazione sia data da un oggetto Java RMI o da uno CORBA scritto in C++.

RMI-IIOP offre in modo del tutto trasparente l'interazione fra Java RMI e CORBA. Il motivo per cui di recente se ne parla sempre più spesso è dovuto al suo utilizzo nell'ambito EJB. Molti application server infatti implementano i bean in esecuzione all'interno del container, con oggetti CORBA, piuttosto che con oggetti Java. Questo, sebbene non abbia nessuna ripercussione sui bean e neppure sul client — gli oggetti CORBA sono creati al momento del deploy — permette un sostanziale aumento delle performance, nonché l'interazione di EJB con tecnologie differenti.

Bibliografia e risorse

[SER] *Java Object Serialization*
http://java.sun.com/j2se/1.3/docs/guide/serialization/

[SERTUT] Tutorial Java, sezione dedicata alla serializzazione
http://java.sun.com/docs/books/tutorial/essential/io/serialization.html

[SERFAQ] FAQ dedicata alla serializzazione
http://java.sun.com/products/jdk/serialization/faq/

[RMI] Home page di RMI
http://java.sun.com/products/jdk/rmi/

[GOF] ERICH GAMM – RICHARD HELM – RALPH JOHNSON – JOHN VLISSIDES, *Design Pattern*, Addison Wesley.

[JPATTERN] JAMES W. COOPER, *Java Design Pattern, A tutorial*, Addison Wesley

[MBPATTERN] *Corso pratico di pattern*, in MokaByte n. 26 (gennaio 1999) e successivi
www.mokabyte.it/1999/01

[RMI-IIOP] Java RMI over IIOP Technology: Documentation Home Page
http://java.sun.com/j2se/1.4.2/docs/guide/rmi-iiop/

[JIDL] LORENZO BETTINI, *Java + IDL = CORBA*, MokaByte 22, settembre 1998
www.mokabyte.it/1998/09

Capitolo 2

Java e i database

NICOLA VENDITTI

Introduzione

Se il materiale su JDBC, le librerie di Java per i database, è di per sé sufficiente per un intero libro, e si farà pertanto uno sforzo sacrificando qualche contenuto per poter adattare il tutto a un singolo capitolo, quello relativo ai database è semplicemente sterminato, Quindi parleremo di database quel tanto che serve per capire in quale area JDBC si posiziona (database relazionali) e quali sono gli aspetti dei database che un programmatore JDBC deve conoscere bene, tanto per evitare inefficienze del codice, che spesso si trasformano in problemi per il database, quanto per trarre tutti i vantaggi possibili dal database stesso come strumento di manipolazione dei dati, evitando di creare codice per fare quello che il database sa già fare (infinitamente meglio).

Il capitolo è organizzato come segue:

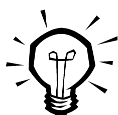
- Una prima parte molto discorsiva sui database allo scopo di circoscrivere il campo di applicazione di JDBC e di dare un piccolo sguardo alle sue alternative.

- Una introduzione a JDBC mirata a fare in modo che il lettore possa da subito scrivere e provare il codice per avere un rapido riscontro a quanto viene illustrato.

- Una sezione dedicata ai concetti di base, come il JDBC URL, la gestione degli errori, i metadati, che serve ad approfondire aspetti particolari di JDBC.

- Una parte più approfondita sul modello delle guide di riferimento, dove, seppure in modo non esaustivo, si espongono in dettaglio almeno la parte e le funzionalità più importanti di JDBC.

- La parte finale del capitolo introduce alcune specializzazioni di JDBC, al di fuori della libreria standard, che possono essere utili nel caso di applicazioni avanzate (transazioni distribuite, rowset, etc.)



Al tempo della redazione di questo capitolo per la prima edizione del libro, JDBC versione 2 era stato da non molto rilasciato. Per cui l'Autore decise di tenere per JDBC v2 una sezione a sé. In questa edizione si conserva la stessa suddivisione più per la convenienza di trattare separatamente le non poche funzionalità aggiuntive di JDBC v2. Per chi ha iniziato a programmare in JDBC a partire dalla versione 2 questa differenza non ha più molto significato.

Che cosa sono i database?

Ogni giorno nel mondo vengono scambiati inimmaginabili volumi di dati e quasi in ogni parte del mondo le informazioni vengono recuperate, elaborate, trasformate, accresciute, aggiornate e infine re-immagazzinate.

I database sono il centro vitale di questo movimento: essi ci permettono non solo di archiviare i dati e le informazioni raccolte nei vari campi dell'attività economica, scientifica e così via, ma garantiscono anche la sicurezza e l'integrità dei dati stessi, così come la possibilità di poterli recuperare in ogni momento nel modo più efficiente e rapido possibile.

Da quando l'informatica ha conquistato un ruolo fondamentale nei vari campi dell'attività umana, è emersa la necessità di avere dei sistemi capaci di gestire in modo avanzato i dati e le informazioni. I DBMS (DataBase Management System) sono la risposta più significativa a questa esigenza. Per DBMS si intende un sistema costituito essenzialmente dal database vero e proprio e, soprattutto, dal software per gestire tutte le operazioni che ordinariamente si fanno su un database, dall'archiviazione all'aggiornamento, fino al backup, al mirroring e così via.

Risale agli inizi degli anni Ottanta la comparsa sul mercato software dei primi DBMS. Si trattava per lo più di sistemi che usavano il file system del sistema operativo che li ospitava come repository per i dati e delle librerie C per accedere ad essi da parte dei programmi client.

Pioniere nel campo della ricerca orientata ai database è stata la IBM che, prima ancora di altri grandi database vendor come Oracle e Informix, si trovò ad affrontare la necessità di fornire sui propri sistemi mainframe, allora già largamente diffusi, del software capace di gestire l'archiviazione dei dati. Originariamente, e in parte anche adesso, era il file che veniva utilizzato direttamente come unità di storage per i database. Così la ricerca si orientò allo studio di un metodo di organizzazione e strutturazione dello spazio nei file per un archiviazione ottimale e un accesso efficiente ai dati. Un risultato tuttora popolare di questa ricerca fu l'ISAM (Indexed Sequential Access Method).

Il concetto di tabella divenne popolare, insieme al modello relazionale, agli inizi degli anni Settanta grazie a Codd (un ricercatore di IBM), che gettò le basi di un approccio teorico ancora largamente utilizzato in questo settore.

Con la comparsa di nuovi protagonisti nel campo dei database, sorse l'esigenza di avere un linguaggio comune per l'accesso ai dati, visto che ognuno disponeva di una propria libreria (ad esempio Informix, nelle primissime versioni del suo database, forniva una libreria detta C-ISAM).

Anche in questo la IBM fu protagonista, e finì per diffondersi un suo linguaggio chiamato SQL (Structured Query Language), oggi molto popolare: da allora non ha subito modifiche sostanziali, ma solo aggiunte.

L'SQL fu derivato a sua volta da un altro linguaggio sperimentale chiamato SEQUEL creato per un sistema che si chiamava System R.

La standardizzazione dell'SQL voluta da ISO e ANSI risale al 1986. Una successiva standardizzazione, nel 1992, introduce nuovi e interessanti elementi senza cambiare la struttura; a questa versione dell'SQL ci si riferisce come SQL-92. L'ultima versione di SQL è SQL99, ovvero SQL3, che introduce il supporto per gli oggetti.

Prima di passare al punto di vista del client di database, e quindi a Java e JDBC, ecco una breve panoramica dei tipi di database esistenti sul mercato. Il modo più comune per classificare i database è quello di discriminarli in base al modello di organizzazione dei dati che utilizzano al loro interno. Usando questo metodo una possibile suddivisione dei database potrebbe essere quella che segue.

Relazionali

Sono i database più diffusi e si basano sul modello relazionale: prevedono quindi un'organizzazione dei dati concettualmente descrivibile in termini di entità e relazioni tra entità; l'accesso e la manipolazione dei dati viene fatto tramite il linguaggio SQL.

Esempi di database relazionali sono: Oracle 9i, Informix Dynamic Server, DB2 Universal Database.

Dimensionali

Sono una specializzazione dei primi per il data warehouse. Si utilizza, di base, il modello relazionale ma cambiano i criteri con cui si organizza una base di dati: si dice tecnicamente che non vale il principio della normalizzazione. Per accedere e utilizzare le caratteristiche di questi database si utilizza per lo più una versione estesa dell'SQL. Questi database system vengono comunemente chiamati OLAP (On Line Analytical Processing) per confronto con i precedenti anche conosciuti con il nome di OLTP (On Line Transaction Processing).

Esempi: Redbrick, Informix XPS e altri.

Object Oriented

Sono molto più recenti dei primi due. Più che basi di dati sono framework per la persistenza di oggetti applicativi. I linguaggi utilizzati per questi database, riconosciuti e standardizzati peraltro dall'OMG (Object Management Group), sono due: OQL (Object Query Language) per la manipolazione dei dati e ODL (Object Definition Language) per la definizione dello schema.

Esempi: Jasmine di CA.

Object Relational

Rappresentano una via di mezzo tra i database relazionali puri e i database OO anche se più esattamente possono essere considerati dei relazionali con estensioni di supporto per la tecnologia Object Oriented.

Per accedere a questi database si utilizza ancora l'SQL, ed è lo stesso SQL, attraverso alcune estensioni proprietarie, a permettere l'accesso alle caratteristiche a oggetti di questi database.

Esempi: Oracle 9i, Informix Dynamic Server.

Accesso ai database: il punto di vista client

Con l'evoluzione dei database sono cambiati i meccanismi con cui le applicazioni accedono ai dati. Originariamente, ogni database prevedeva proprie librerie C. Se quindi un'applicazione doveva per qualche ragione utilizzare un nuovo database, occorreva riscrivere tutto il codice di gestione dei dati.

Per rimediare a ciò fu creato uno standard a "livello di chiamata" detto appunto Call Level Interface (CLI) proposto da X/Open. Fu cioè definita la sequenza di chiamate che l'applicazione (per lo più scritta in C) doveva seguire per accedere in modo corretto alle informazioni. I produttori di database hanno iniziato a fornire questo set di librerie in aggiunta alle proprie originarie, che continuavano a essere la base.

JDBC è un modello anch'esso basato sullo standard CLI per l'accesso alle basi di dati. Per il fatto di utilizzare un linguaggio ad oggetti, JDBC semplifica ancora di più il codice necessario per le operazioni sul database. Si pensi infatti che con solo 4 righe di codice è possibile: caricare il driver adatto alla base di dati che voglio interrogare, ottenere la connessione, creare lo statement e recuperare il result set esplicitando la query.

I problemi che occorre affrontare quando si scrive una libreria per l'accesso ai database sono diversi: si deve garantire, dal lato client, una coerenza logica il più possibile vicina alla filosofia del linguaggio che l'applicazione usa, adattandosi ai metodi tra loro molto differenti che i DBMS utilizzano per processare le richieste dei client; si deve fornire poi una specifica per la scrittura e l'implementazione dei driver e non da ultimo convincere i produttori di database della opportunità di scrivere driver per questa nuova interfaccia. Pertanto il risultato di semplicità e universalità di JDBC è tanto più apprezzabile.

JDBC è una interfaccia a oggetti per l'esecuzione di comandi SQL: è bene sottolineare quindi che il vero medium per la comunicazione con il database rimane il linguaggio SQL e lo stesso vale per le altre tecnologie a oggetti concorrenti di Java come DAO di Microsoft.

In questo senso JDBC altera il paradigma Object Oriented (OO), dove non esiste il concetto di dato semplice né tantomeno quello di riga e di tabella. Per i linguaggi OO occorrerebbe, più che una repository di dati, come nei database tradizionali, un framework per la persistenza degli oggetti: tutto quello che l'applicazione dovrebbe fare sarebbe solo di indicare un'astratta repository da cui recuperare un oggetto che rappresenta un certo elemento di realtà e in cui immagazzinare o più precisamente rendere persistenti gli oggetti nel loro stato applicativo.

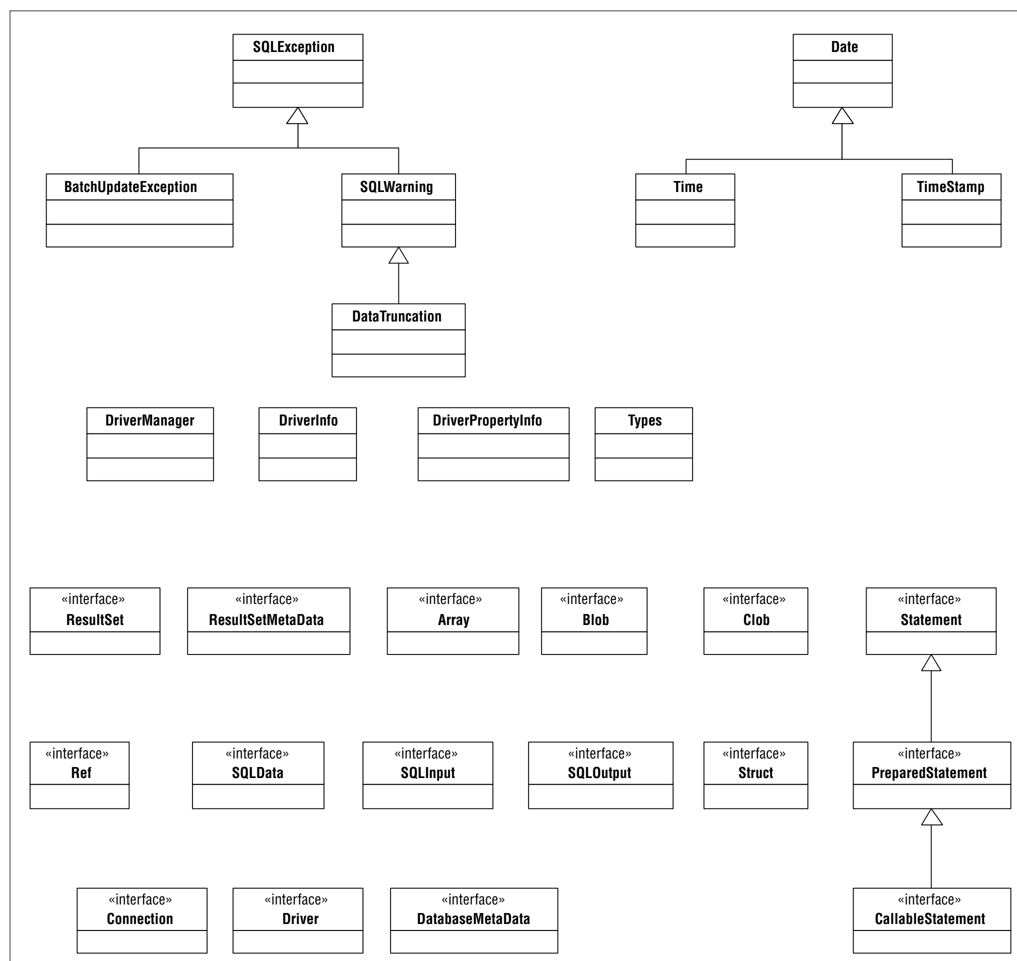
A parte queste obiezioni, a cui in parte cerca di rimediare JDO (Java Data Objects) di Sun, JDBC è molto flessibile, anche grazie al fatto che è stato progettato basandosi sul concetto di

interfaccia e non di oggetto: tutte le operazioni necessarie per accedere ai dati vengono fatte attraverso i metodi esposti da interfacce (che il driver JDBC implementa) e non occorre istanziare alcun oggetto.

Introduzione a JDBC

Per dare subito un'idea concreta di come è articolata l'API di JDBC riportiamo lo schema UML della versione attuale della libreria:

Figura 2.1 — *Lo schema dell'API JDBC con classi e interfacce.*



La più semplice e comune operazione sui database sono le query SQL. I passi da compiere per l'esecuzione di una query con l'API JDBC sono principalmente i seguenti:

1. Caricamento driver JDBC e connessione al DB

Per prima di ogni cosa occorre caricare il driver che gestisce la nostra base dati, in modo esplicito o implicito. Ecco un esempio di caricamento esplicito della classe di un driver, per caricare il driver JDBC tipo 4 di Informix IBM:

```
Class.forName("com.informix.jdbc.IfxDriver");
```

Il driver (o meglio, la classe che lo rappresenta) sarà d'ora in poi la nostra interfaccia con il DB.

La seconda operazione da fare è la connessione al DB, univocamente individuato dalla stringa o URL JDBC di connessione. Al termine di questa operazione si dispone di un oggetto di tipo Connection che rappresenta la connessione stessa. Per la connessione l'applicazione si affida al Driver Manager; l'applicazione semplicemente richiede una connessione specificando l'URL del database a cui desidera connettersi.

```
Connection conn = DriverManager.getConnection(
"jdbc:informix://db.mokabyte.it:5325:INFORMIXSERVER=ol_ifx;DATABASENAME=mokadb", "ifxuser", "ifxpasswd");
```

Sarà proprio il driver manager a recuperare il driver giusto interrogandoli uno per uno con la seguente domanda: gestisci l'URL JDBC jdbc:informix://...? Il primo driver che risponde true a questa domanda sarà selezionato dal driver manager e da questo punto in poi responsabile di tutte le operazioni sul database. Il passo precedente in cui abbiamo caricato la classe del driver nella virtual machine era quindi necessario perché il driver manager possa trovare il driver giusto per gestire il tipo di URL JDBC da noi richiesto. Il JDBC URL è quindi strettamente legato al tipo di driver e di database, e specifica le informazioni necessarie per la connessione. In questo caso per connettersi a un database Informix IBM occorre il nome dell'istanza e il nome del database, oltre che il nome DNS del server su cui gira e la porta TCP/IP.

2. Creazione dell'oggetto Statement

Creata la connessione al DB si è in possesso di un oggetto che la rappresenta. Da esso è possibile poi ottenere uno dei tre diversi tipi di interfacce che effettivamente permettono di sottomettere istruzioni SQL al DB. Queste sono: Statement, PreparedStatement e CallableStatement. Ecco un frammento di codice che recupera lo statement

```
Statement stmt = conn.createStatement();
```

Come vedremo, si possono recuperare versioni più specializzate per lavorare per esempio con le stored procedure; l'esempio sopra riportato è però sufficiente a dare l'idea.

3. Esecuzione della query

Una volta creato uno statement non resta che eseguire la query SQL. Ecco un esempio:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM lettori");
```

Questa istruzione interroga, attraverso lo statement, una tabella "lettori" (diciamo i lettori di MokaByte), e mette i risultati in un oggetto ResultSet contenente i dati recuperati in forma di un array di record.

4. Elaborazione dei risultati

L'oggetto ResultSet (o per meglio dire l'interfaccia) ha una serie di metodi che permettono di scorrere l'array dei risultati. In particolare attraverso il suo metodo next() è possibile percorrere tali record dal primo all'ultimo, mentre i metodi getXXX() permettono di accedere ai singoli campi del record.

Come detto all'inizio, quando si costruisce una libreria per i database occorre che questa si conformi al modello di accesso ai dati utilizzato del medesimo. Questo è un esempio evidente. In questo caso sembra di avere a disposizione un oggetto che contiene al suo interno il set di dati recuperato, ma non è così. In realtà la query ha solo creato un cursore che punta inizialmente alla prima riga del set di risultati ritornati: il metodo next() serve per spostare il cursore in avanti e i metodi getXXX() si utilizzano per spostare effettivamente i dati dal server al client. Anche se si parla di array di record, quindi, esso non è da intendersi come una struttura di dati locale, ma come un riferimento a un'area di dati sul server di database. Si tornerà su questa differenza a proposito dei RowSets.

Il meccanismo per la scansione dei risultati è mostrato nel codice che segue.

```
String nome = null;
String cognome=null;
int eta;
while(rs.next()) {
...
nome = rs.getString(1);  // colonna 1 contiene il nome
        // si può anche indicare la colonna della tabella
// per nome, ad esempio rs.getString("NomeUtente");
cognome = rs.getString(2); // colonna 2 contiene il cognome
eta = rs.getInt(2);      // colonna 3 contiene l'età
        // per un Integer si può anche invocare il metodo getString,
        // ottenendo una conversione automatica;
...
}
```

5. Rilascio delle risorse utilizzate

Una volta completate le operazioni sul database è molto importante rilasciare le risorse acquisite. Ciò comporta per lo meno la chiusura degli oggetti ResultSet, Statement e Connection, nell'ordine inverso alla loro apertura. Se occorre, è disponibile un metodo su tali oggetti per verificare se l'operazione è già stata effettuata, isClose(), che però dice solo che il metodo close() sull'oggetto è stato già chiamato, e non se la connessione è attiva o meno.

Ecco tradotta in un piccolo esempio di programma (che corrisponde al programma demo SempliceQueryOra) la sequenza di passi appena illustrata.

```
…
  try {
  // PASSO 1
  // Caricamento esplicito del driver JDBC per il tipo di sorgente
  // di dati che mi interessa: in questo caso una fonte ODBC
    Class.forName("oracle.jdbc.driver.OracleDriver");

    // PASSO 2
    // Creazione della connessione al database (tramite il Driver Manager)
    Connection con
    = DriverManager.getConnection("jdbc:oracle:thin:@dbserver.mokabyte.it:1521:mokadb", "moka", "corretto");

    // PASSO 3
    // Creazione dello statement, l'oggetto che utilizzo per
    // spedire i comandi SQL al database
    Statement st = con.createStatement();

    // PASSO 4
    // Eseguo la query o un altro comando SQL attraverso lo statement
    // e recupero i risultati attraverso l'interfaccia ResultSet
    ResultSet rs = st.executeQuery("select nome, cognome, email from lettori");

    // PASSO 5
    // Scandisco il result set per la visualizzazione
    // delle informazioni recuperate
    while (rs.next())
    {
      System.out.println("Nome: " + rs.getString(1));
      System.out.println("Cognome: " + rs.getString(2));
      System.out.println("Email: " + rs.getString(3));
      System.out.println(" ");
    }

    // PASSO 6
    // Rilascio le risorse utilizzate
    rs.close();  // result set
    st.close();  // statement
    con.close(); // connessione
  }
   catch(ClassNotFoundException cnfe) {
```

```
      // Class.forName può lanciare questa eccezione!
        System.err.println("Attenzione classe non trovata" + cnfe.getMessage());
    }
     catch(SQLException sqle) {
      System.err.println("Attenzione errore SQL" + sqle.getMessage());
    }
```

L'esempio riporta la sequenza di passi essenziali per eseguire una query SQL. Ovviamente JDBC permette tutte le operazioni SQL di manipolazione dei dati (comandi DDL ovvero Data Definition Language e DML ovvero Data Manipulation Language di SQL).

Nel seguito viene riportato il codice del programma di esempio UpdateShell, una piccola shell da linea di comando che permette di inserire e inviare al database comandi di tipo DDL e DML (tranne query):

```
out.println("\nInserire i comandi DML/DDL su una sola riga e premere invio.");
out.println("Es.: create table testtab (a integer);\n");

int updret;
while(true){
  out.print("UpdateShell>> ");
  out.flush();

  // prende SQL
  while((sql = in.readLine()).equals(""))
    out.print("\nUpdateShell>> ");;

  if(sql.equals("exit") || sql.equals("quit"))
    break;

  // rimuove il ';' finale, per evitare errore da parte
  // del driver JDBC Oracle.
  // Usando il driver JDBC Informix questo problema non c'è.
  if(sql.charAt(sql.length() - 1) == ';')
    sql = sql.substring(0, sql.length() - 1);

  DriverManager.println(">> Esecuzione operazione <" + sql + ">");

  // Eseguo comando di aggiornamento
  try{
    updret = stmt.executeUpdate(sql);
    out.println("Aggiornato/i " + updret + " records.\n");
    out.flush();
  }catch(SQLException sqle) {
```

```
        System.err.println("Mokaproblem: executeUpdate: "
          + sqle.getMessage());
        continue;
      }
    }
```

Il programma utilizza il metodo Statement.executeUpdate() sia per comandi tipo CREATE TABLE prova(a INT) detti Data Definition Language (DDL) sia per comandi come UPDATE lettori SET non_pubblico ='S' e DELETE FROM sottoscrizioni WHERE iduser = 119 che sono detti di Data Manipulation Language (DML).

Eseguendo il programma come segue:

```
D:\dev>java -Djdbc.drivers
=oracle.jdbc.driver.OracleDriver it.mokabook.jdbc.UpdateShell jdbc:oracle:thin:@dbserver.mokabyte.it:1521:mokadb moka corretto
```

si ottiene ad esempio:

```
UpdateShell>> create table tab (a integer);
Aggiornato/i 0 records.

UpdateShell>> insert into tab values (1);
Aggiornato/i 1 records.

UpdateShell>> insert into tab values (3);
Aggiornato/i 1 records.

UpdateShell>> delete from tab where a <=3;
Aggiornato/i 2 records.
```

Il codice del programma è abbastanza semplice:

```
…
int updret;
  while(true){
    out.print("UpdateShell>> ");
    out.flush();

    // prende SQL
    while((sql = in.readLine()).equals(""))
      out.print("\nUpdateShell>> ");;

    if(sql.equals("exit") || sql.equals("quit"))
      break;
```

```
// rimuove il ';' finale, per evitare errore da parte
// del driver JDBC Oracle.
// Usando il driver JDBC Informix questo problema non c'è.
if(sql.charAt(sql.length() - 1) == ';')
   sql = sql.substring(0, sql.length() - 1);

DriverManager.println(">> Esecuzione operazione <" + sql + ">"); // log...

// Eseguo comando di aggiornamento
try{
   updret = stmt.executeUpdate(sql);
   out.println("Aggiornato/i " + updret + " records.\n");
   out.flush();
}catch(SQLException sqle){
   System.err.println("Mokaproblem: executeUpdate: " + sqle.getMessage());
   continue;
}
…
```

Il metodo Statement.executeUpdate(String) restituisce 0 per comandi DDL oppure un row count ovvero il numero di righe modificate da un comandi di UPDATE/DELETE/INSERT.

Esistono anche versioni più avanzate di Statement.executeUpdate, quali ad esempio Statement.executeUpdate(String, String[]), che permette di recuperare le chiavi autogenerate per le colonne auto-incrementanti, ma con questo si entra già nello specifico del database utilizzato dal programma.

Concetti di base di JDBC

Nella precedente sezione abbiamo fatto le prime esperienze di programmazione con JDBC, imparando a connettersi, ricercare, e modificare i dati, adesso ci soffermiamo su alcuni concetti di base che è bene comprendere in modo più approfondito, prima di proseguire.

Convenzioni per il JDBC URL Naming

Perché un'applicazione Java possa accedere a una fonte di dati occorre che essa specifichi al driver manager le informazioni necessarie per localizzarla. Tutte queste informazioni devono essere rese disponibili al momento della connessione in formati prestabiliti. La soluzione scelta consiste nell'impacchettare tutti questi dati in una stringa, detta a volte stringa di connessione, secondo uno schema definito dallo stesso JDBC e molto simile agli URL del Web.

La generica struttura è la seguente:

```
jdbc:<sub-protocollo>:<nome-dominio>
```

Nella sintassi URL il primo termine indica il protocollo da utilizzare per la gestione della risorsa individuata, che in questo caso è ovviamente jdbc.

Il sottoprotocollo è un identificatore associato a un certo DBMS (p.e.: oracle, informix). Per nome-dominio si intende, una stringa che permette di localizzare univocamente la sorgente dati ed è anche questa specifica del DBMS (di solito comprende almeno il nome DNS della macchina che ospita il database e la porta TCP/IP). Nel caso più comune di utilizzo del driver JDBC bridge jdbc-odbc, un URL potrebbe essere il seguente:

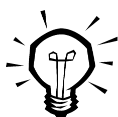
```
jdbc:odbc://www.infomedia.it/dblettori
```

In questo modo si indica che la risorsa che vogliamo raggiungere attraverso JDBC e il driver bridged JDBC-ODBC, si chiama dblettori. Come detto l'interpretazione del domain-name dipende dal driver. Nel caso del bridge il nome dblettori rappresenta il nome di un DSN (Data Source Name).

Un esempio più elaborato può essere quello del thin driver di Oracle:

```
jdbc:oracle:thin:@db.mokabyte.it:1621:mokadb
```

La parte nome-dominio dell'URL, che inizia con @, specifica la macchina (db.mokabyte.it), la porta (1621) e l'Oracle SID (mokadb), che serve a specificare il database vero e proprio (ci possono infatti essere più istanze attive di Oracle, ognuna associata a un database identificato appunto dal SID).



Oracle mette a disposizione degli sviluppatori due tipi di driver: *thin* e *OCI*. L'OCI driver si distingue dal primo, il thin, che i programmatori devono privilegiare, essenzialmente perché effettua chiamate a funzioni native (OCI, Oracle Call-level Interface, è appunto il layer nativo del database). Quest'ultimo viene utilizzato solo in casi speciali, per l'accesso a funzionalità avanzate del database quali il Transparent Fail-Over, o perché si vuole il massimo delle performance, soprattutto nella manipolazione dei LOB.

Come si utilizza un JDBC URL? Ecco l'esempio di gran lunga più comune:

```
String mokadbUrl = "jdbc:oracle:thin:@db.mokabyte.it:1621:mokadb"
Connection conn = DriverManager.getConnection(mokadbURL, user, passwd);
// il resto del programma utilizza l'oggetto conn per accedere al db
...
```

L'esempio mostra come le credenziali utente vengano passate separatamente dal JDBC URL. Alcuni driver permettono di specificare tutto nell'URL, anche se non è una caratteristica mol-

to utilizzata. Ecco un esempio di connessione con il driver OCI di Oracle, in cui si specificano nome utente e password (giovanni/puliti):

```
String mokadbUrl = "jdbc:oracle:oci8:giovanni/puliti@db.mokabyte.it"
Connection conn = DriverManager.getConnection(mokadbURL);
```

Driver e DriverManager

Sfogliando la libreria di JDBC, contennuta nel package java.sql.*, si vede come tutti i componenti principali, come Connection, Statement, ResultSet, DatabaseMetaData, siano interfacce e non classi. DriverManager, al contrario, è una classe, ed è in effetti la classe più importante della libreria. Vediamo perché.

Gestione dei driver

JDBC dà la possibilità di collegarsi a diversi tipi di database individuati da diversi JDBC URL. Serve quindi un meccanismo che permetta ai driver di database di registrarsi nella virtual machine, in modo che quando il programma invoca la seguente riga di codice:

```
Connection conn = DriverManager.getConnection("jdbc:oracle:thin:@db.mokabyte.it:1621:mokadb", user, passwd);
```

il driver manager possa recuperalo e gestire le operazioni su database.

Serve altresì un meccanismo che indichi al driver manager, nel caso vi siano più driver registrati, qual'è quello che gestisce il tipo di connessioni jdbc:oracle:thin o in generale che gestisca il JDBC URL specificato dal programma.

Per la gestione dei driver il driver manager mette a disposizione i metodi, nota bene, statici:

```
static void registerDriver(Driver driver);
static Driver getDriver(String jdbcURL);
static Enumeration getDrivers();
static void deregisterDriver(Driver driver);
```

Il primo, DriverManager.registerDriver, viene invocato dalle classi driver di database per registrarsi, quando vengono caricate nella JVM. Il secondo invece, permette l'associazione JDBC URL con il driver specifico di cui si è parlato.

La registrazione di un driver può avvenire in due modi. Il primo è per via programmatica, usando una delle seguenti istruzioni:

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

oppure

```
DriverManager.registerDriver( new oracle.jdbc.driver.OracleDriver() );
```

La prima viene detta registrazione implicita, perché la classe OracleDriver ha un blocco di inizializzazione statico che in modo trasparente per il programmatore si registra presso il driver manager. La seconda viene detta registrazione esplicita, per il motivo opposto. La Oracle per esempio, consiglia di utilizzare il secondo per evitare problemi di compatibilità con alcune VM.

Il secondo modo utilizza la system property jdbc.drivers. Questa proprietà può essere automaticamente impostata quando si invoca il programma. Eseguendo il programma di esempio DriverDemo nel seguente modo:

```
D:\esempi>java -cp %CLASSPATH% -Djdbc.drivers
=oracle.jdbc.driver.OracleDriver it.mokabyte.jdbc.DriverDemo jdbc:oracle:thin:@db.mokabyte.it:1521:mokadb
```

si ottiene il seguente output:

```
Elenco dei driver caricati:

    ----------------------------
    Driver #0 oracle.jdbc.driver.OracleDriver@50169
    ----------------------------
    JDBC compliant?: yes
    Versione: 1.0
```

Mentre invocandolo nel seguente modo:

```
D:\esempi>java -cp %CLASSPATH% -Djdbc.drivers
=sun.jdbc.odbc.JdbcOdbcDriver it.mokabyte.jdbc.DriverDemo jdbc:odbc:mokasource
```

si ottiene:

```
Elenco dei driver caricati:
    ----------------------------
    Driver #0 sun.jdbc.odbc.JdbcOdbcDriver@53c015
    ----------------------------
    JDBC compliant?: yes
    Versione: 2.1
    Driver Property Info:
    Nome: UID
    Description: User Name
    Nome: PWD
    Description: Password
    Nome: DBQ
    Description: Service Name
    Nome: DBA
...
```

mokasource in questo caso è un DataSource locale (secondo la terminologia Microsoft) creato dall'Amministratore delle risorse ODBC del sistema Windows.

Connessione e altre funzionalità

L'altra ragione per cui la classe DriverManager è fondamentale è perché permette di creare la connessione al database, che è il punto di partenza per le operazioni SQL. Ottenere la connessione è piuttosto semplice, basta scegliere tra i metodi DriverManager.getConnection quello che meglio si adatta al proprio programma e passare le relative informazioni. Ci sono altre funzionalità del driver manager spesso non considerate, che possono al contrario essere molto utili come ad esempio il logging, che può rivelarsi un ottimo strumento di analisi in caso di problemi. L'esempio DriverManagerDemo, di cui mostriamo solo la parte di codice che attiva il log:

```
// Impostazione stream di log SQL
DriverManager.setLogStream(new PrintStream(new FileOutputStream("DriverManagerDemo.log")));
```

produce il seguente output nel file DriverManagerDemo.log (accorciato per ragioni di spazio):

```
DriverManager.getConnection("jdbc:oracle:thin:@lunarossa.italy.sun.com:1521:sunone")
   trying driver[className=oracle.jdbc.driver.OracleDriver,oracle.jdbc.driver.OracleDriver@50169]
DRVR OPER Enabled logging (moduleMask 0x0fffffff, categoryMask 0x0fffffff)
DRVR DBG1 doDefaultTypes
DRVR DBG1 doDefinesFromTypes
DBAC FUNC DBData.DBData(nItems=10)
DRVR DBG1 After execute: valid_rows=1
DRVR DBG2 defines:  oracle.jdbc.dbaccess.DBDataSet@56a499
…
DRVR OPER OracleStatement.executeQuery(sql)
DRVR DBG1 SQL: "SELECT TABLE_NAME AS Nome FROM ALL_TABLES"
DRVR FUNC OracleStatement.sendBatch()
DRVR FUNC OracleStatement.doExecuteWithTimeout()
DRVR FUNC OracleConnection.nativeSQL(sql)
DRVR DBG1 Input SQL:  "SELECT TABLE_NAME AS Nome FROM ALL_TABLES"
DRVR DBG1 Output SQL: "SELECT TABLE_NAME AS Nome FROM ALL_TABLES"
DBCV FUNC DBConversion.StringToCharBytes(str)
DBCV FUNC DBConversion.stringToAcessCharBytes(str, charset=31)
```

Gestione degli errori

Si prende ora in considerazione un aspetto abbastanza importante, quello della gestione delle eccezioni; importante soprattutto se ci sono problemi con il database. Utilizzando in modo accorto le SQLException e i SQLWarning si possono ottenere molte informazioni su quello che sta realmente accadendo e possibilmente individuare l'errore.

Le SQLException

Negli esempi riportati precedentemente si è potuto osservare che, ogni volta che viene eseguita una istruzione JDBC, può essere intercettata una eccezione di tipo SQLException. Questa classe offre una serie di informazioni relative al tipo di errore verificatosi. Essa deriva dalla più generale java.lang.Exception, la quale a sua volta deriva dalla java.lang.Throwable. Le informazioni contenute nella classe sono le seguenti:

- Il tipo di errore verificato sotto forma di una stringa descrittiva; tale informazione può essere utilizzata come exception message e può essere ricavata per mezzo del metodo getMessage().
- Una proprietà (SQLState) descrivente l'errore in base alle convenzioni dello standard X/Open SQLState. Può essere ottenuta con getSQLState().
- Un codice di errore specifico del produttore del database, che in genere corrisponde al messaggio di errore fornito dal DBMS stesso; getErrorCode() permette la sua lettura.
- Un collegamento al successivo oggetto di tipo SQLException, eccezione che può essere utilizzata se si sono verificati diversi errori. Il metodo getNextException() permette di navigare nella catena di eccezioni riscontrate da un metodo.

Nell'esempio che segue è mostrato come utilizzare le eccezioni e i warning per avere una descrizione completa dell'errore verificatosi.

```
…
static public void main(String[] args){
  try {
    // Carica il driver JDBC per ORACLE
    Class.forName("oracle.jdbc.driver.OracleDriver");

    // Si connette al database con SID=mokadb
    Connection con
     = DriverManager.  getConnection("jdbc:oracle:thin:@dbserver.mokabyte.it:1521:mokadb", "moka", "corretto");

    // Crea lo statement
    Statement st = con.createStatement();

    //esegue una istruzione errata: la tabella Lettttori non esiste
    ResultSet rs = st.executeQuery("select * from Lettttori");

  } catch(ClassNotFoundException cnfe) {
    System.out.println("Classe non trovata" + cnfe.getMessage());
```

```
    } catch(SQLException sqle) {
      System.out.println("Attenzione errore SQL" + "\n");
      while (sqle != null) {
      System.out.println("Messaggio SQL \n" + sqle.getMessage() + "\n");
      System.out.println("SQLState \n" + sqle.getSQLState() + "\n");
        System.out.println("Codice errore del produttore \n" + sqle.getErrorCode()+ "\n");
        System.out.println("Traccia dello StackTrace");
        sqle.printStackTrace(System.out);
        sqle = sqle.getNextException();
        System.out.println("");
      }
    }
}
…
```

Eseguendo questo programma su un database Oracle, in cui non esiste la tabella Lettttori ecco cosa si ottiene:

```
Attenzione errore SQL

Messaggio SQL
ORA-00942: tabella o vista inesistente


SQLState
42000

Codice errore del produttore
942

Traccia dello StackTrace
java.sql.SQLException: ORA-00942: tabella o vista inesistente

  at oracle.jdbc.dbaccess.DBError.throwSqlException(DBError.java:134)
  at oracle.jdbc.ttc7.TTIoer.processError(TTIoer.java:289)
  at oracle.jdbc.ttc7.Oall7.receive(Oall7.java:573)
  at oracle.jdbc.ttc7.TTC7Protocol.doOall7(TTC7Protocol.java:1891)
  at oracle.jdbc.ttc7.TTC7Protocol.parseExecuteDescribe(TTC7Protocol.java:830)
  at oracle.jdbc.driver.OracleStatement.doExecuteQuery(OracleStatement.java:2391)
  at oracle.jdbc.driver.OracleStatement.doExecuteWithTimeout(OracleStatement.java:2672)
  at oracle.jdbc.driver.OracleStatement.executeQuery(OracleStatement.java:572)
  at Eccezioni_e_Warnings.main(Eccezioni_e_Warnings.java:19)
```

I warning

Oltre alle eccezioni alle quali si è abituati in Java, nel caso specifico di JDBC è disponibile anche un altro strumento per la gestione delle situazioni anomale. La differenza che sussiste fra un warning e una eccezione risiede essenzialmente nel fatto che il primo non interrompe il flusso del programma: si definisce infatti come un segnalatore silenzioso di anomalie. Un warning viene scatenato direttamente dal database, in funzione del tipo di errore che si verifica.

Non è possibile generalizzare quando l'uno o l'altro tipo di strumento viene generato, perché ciò dipende dall'implementazione del DB. I warning offrono lo stesso tipo di informazioni delle eccezioni `SQLException`, e possono essere ricavati con metodi del tutto identici rispetto al caso precedente.

Per vedere i warning prodotti dal database (se ve ne sono) durante le operazioni è possibile modificare leggermente l'esempio precedente:

```
…
public class Eccezioni_e_Warnings_2{

 static public void main(String[] args){
  try {
    // Carica il driver JDBC per ORACLE
    Class.forName("oracle.jdbc.driver.OracleDriver");

    // Si connette al database Oracle con SID=mokadb
    Connection con
     = DriverManager.   getConnection("jdbc:oracle:thin:@dbserver.mokabyte.it:1521:mokadb","moka",  "corretto");

    con.commit();
    showSQLWarnings(con.getWarnings());

    // Crea lo statement
    Statement st = con.createStatement();
    showSQLWarnings(con.getWarnings());

    // USER_TABLES è una tabella che esiste sempre
    ResultSet rs = st.executeQuery("select count(*)  as tabs from user_tables");
    rs.next();
    System.err.print("L'utente \"oracle\" ha " + rs.getString("tabs") + " tabelle\n");

    showSQLWarnings(st.getWarnings());
    showSQLWarnings(rs.getWarnings());

  } catch(ClassNotFoundException cnfe) {
    System.out.println("Classe non trovata" + cnfe.getMessage());
```

```
    } catch(SQLException sqle)  {
      System.out.println("Attenzione errore SQL" + "\n");
      while (sqle != null) {
        System.err.println("Messaggio SQL: " + sqle.getMessage() + "\n");
        System.err.println("SQLState: " + sqle.getSQLState() + "\n");
        System.err.println("Codice errore del produttore: " + sqle.getErrorCode()+ "\n");
        System.err.println("Traccia dello StackTrace");
        sqle.printStackTrace(System.out);
        sqle = sqle.getNextException();
        System.err.println("");
        }
      }
    }
    // showSQLWarnings: visualizza i warnings generati da una istruzione JDBC
    static public int showSQLWarnings(SQLWarning first){
    int warncount = 0;

    // scandisce la catena dei warnings prodotta dall'istruzione SQL
    SQLWarning w = null;
    for(w = first; w != null; w = w.getNextWarning()){
      System.err.println("\tWarning: Messaggio SQL: " + w.getMessage() + "\n");
      System.err.println("\tWarning: SQLState: " + w.getSQLState() + "\n");
      System.err.println("\tWarning: Codice produttore: " + w.getErrorCode()+ "\n");
      warncount++;
    }
    // restituisce il numero dei warnings instanziati
    return warncount;
    }
  }
```

Il metodo showSQLWarnings() semplifica la visualizzazione dei warning generati dal database.

I metadati

JDBC permette quello che la documentazione di Sun chiama "accesso dinamico ai database". Il nome non indica molto, ma si tratta della possibilità di accedere a un database e ricavarne informazioni sulla sua struttura interna (tabelle, relazioni, sinonimi, link, trigger, etc.) senza saperne nulla a priori. In questo modo si può, per esempio, interrogare una tabella utenti senza sapere in anticipo quali e quante sono le colonne che la compongono.

Ciò è possibile perché tutti i maggiori DBMS hanno delle tabelle interne dette "dizionari" o "cataloghi" che contengono metainformazioni circa la struttura interna dai database. Per fare un esempio, quando si crea un database Informix, prima ancora che vengano create le tabelle dagli utenti abilitati, ne viene automaticamente creata qualche decina durante il normale processo di

creazione del database. Se si hanno i privilegi e si esegue il comando "SELECT * FROM systables" si può aver l'elenco di tali tabelle che hanno nomi a volte molto espliciti: sysindexes, systriggers, syscolumns, etc. Come si può intuire ognuna di queste tabelle rappresenta il catalogo per il relativo oggetto: ad esempio syscolumns contiene tutte le colonne di tutte le tabelle create dall'utente, systriggers contiene tutti i trigger creati su tutti gli oggetti che supportano gli eventi e così via.

L'esempio più semplice di utilizzo delle classi di metadati è il seguente, tratto dall'esempio QueryShell, una piccola shell che permette all'utente di inserire query SQL arbitrarie, eseguirle e visualizzarne il risultato:

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(line);
ResultSetMetaData rsmd = rs.getMetaData();
int cols = rsmd.getColumnCount();

// stampa le colonne
// Nota: le colonne hanno indice che parte da 1
for (int i = 1; i <= cols; ++i)
{
  out.print(rsmd.getColumnName(i));
  if( i < cols )
    out.print(" ");
}

// stampa le righe del result set
while(rs.next())
{
  for (int i = 1; i <= cols; ++i)
  {
    out.print(rs.getString(i));
    if( i < cols )
      out.print("|");
  }
  out.println("");
}
```

La stringa line contiene una query SQL inserita dall'utente. Il programma non sa quindi a priori quale tabella viene interrogata, né la sua struttura (colonne). Se non ci fosse la classe ResultSetMetaData con i relativi metodi getColumnCount() e getColumnName() il programma non potrebbe stampare l'intestazione dell'output con il nome delle colonne che sono interrogate (nell'ordine e con l'alias eventualmente specificato nella query).

Il programma di esempio ResultSetMetaDataDemo.java è semplice ma interessante, perché permette di inserire dal prompt una qualsiasi query SQL, eseguirla e visualizzare quasi tutti i metadati possibili. Si veda ad esempio la seguente query (volutamente bizzarra):

```
ResultSetMetaDataDemo>>
SELECT iduser AS ID, nome || ' ' || cognome AS NomeCompleto, data, 'Anni: ', 1-2 AS DIFF,
sin(iduser), (select table_name from user_tables where table_name = 'lettori')  FROM lettori
Colonna #1 ID
        Nome fisico: ID
        Nome classe: java.math.BigDecimal
        Nome display size: 22
        Nome label: ID
        Tipo colonna: 2
        Nome tipo colonna: NUMBER
        Sola lettura?: false
        Ricercabile?: true
        E' scrivibile?: true
        E' definitivamente scrivibile?: false
        Nome schema:
        Nome tabella?:
        Nome catalogo:
        Colonna autoincrementata?: false
        Case Sensitive?: false
        Nullable?: 0
Colonna #2 NOMECOMPLETO
        Nome fisico: NOMECOMPLETO
        Nome classe: java.lang.String
        Nome display size: 71
        Nome label: NOMECOMPLETO
        Tipo colonna: 12
        Nome tipo colonna: VARCHAR2
        Sola lettura?: false
        Ricercabile?: true
        E' scrivibile?: true
        E' definitivamente scrivibile?: false
        Nome schema:
        Nome tabella?:
        Nome catalogo:
        Colonna autoincrementata?: false
        Case Sensitive?: true
        Nullable?: 1
Colonna #3 DATA
        Nome fisico: DATA
        Nome classe: java.sql.Timestamp
        Nome display size: 7
        Nome label: DATA
        Tipo colonna: 91
```

```
        Nome tipo colonna: DATE
        Sola lettura?: false
        Ricercabile?: true
        E' scrivibile?: true
        E' definitivamente scrivibile?: false
        Nome schema:
        Nome tabella?:
        Nome catalogo:
        Colonna autoincrementata?: false
        Case Sensitive?: false
        Nullable?: 1
Colonna #4 ANNI:
        Nome fisico: ANNI:
        Nome classe: java.lang.String
        Nome display size: 6
        Nome label: ANNI:
        Tipo colonna: 1
        Nome tipo colonna: CHAR
        Sola lettura?: false
        Ricercabile?: true
        E' scrivibile?: true
        E' definitivamente scrivibile?: false
        Nome schema:
        Nome tabella?:
        Nome catalogo:
        Colonna autoincrementata?: false
        Case Sensitive?: true
        Nullable?: 1
Colonna #5 DIFF
        Nome fisico: DIFF
        Nome classe: java.math.BigDecimal
        Nome display size: 22
        Nome label: DIFF
        Tipo colonna: 2
        Nome tipo colonna: NUMBER
        Sola lettura?: false
        Ricercabile?: true
        E' scrivibile?: true
        E' definitivamente scrivibile?: false
        Nome schema:
        Nome tabella?:
        Nome catalogo:
        Colonna autoincrementata?: false
```

```
        Case Sensitive?: false
        Nullable?: 1
    Colonna #6 SIN(IDUSER)
        Nome fisico: SIN(IDUSER)
        Nome classe: java.math.BigDecimal
        Nome display size: 22
        Nome label: SIN(IDUSER)
        Tipo colonna: 2
        Nome tipo colonna: NUMBER
        Sola lettura?: false
        Ricercabile?: true
        E' scrivibile?: true
        E' definitivamente scrivibile?: false
        Nome schema:
        Nome tabella?:
        Nome catalogo:
        Colonna autoincrementata?: false
        Case Sensitive?: false
        Nullable?: 1
    Colonna #7 (SELECTTABLE_NAMEFROMUSER_TABLESWHERETABLE_NAME=LETTORI)
        Nome fisico: (SELECTTABLE_NAMEFROMUSER_TABLESWHERETABLE_NAME=LETTORI)
        Nome classe: java.lang.String
        Nome display size: 30
        Nome label: (SELECTTABLE_NAMEFROMUSER_TABLESWHERETABLE_NAME=LETTORI)
        Tipo colonna: 12
        Nome tipo colonna: VARCHAR2
        Sola lettura?: false
        Ricercabile?: true
        E' scrivibile?: true
        E' definitivamente scrivibile?: false
        Nome schema:
        Nome tabella?:
        Nome catalogo:
        Colonna autoincrementata?: false
        Case Sensitive?: true
        Nullable?: 1
```

Notare ad esempio che la colonna iduser, che nel nostro database di prova abbiamo definito con il constraint NOT NULL, riporti correttamente Nullable=0. Il driver Oracle sembra non fare differenza tra la label, definita nella query attraverso la parola chiave AS, per esempio ID, e il nome fisico della tabella. La classe ResultSetMetaData permette di avere metadati di un result set. Se occorrono informazioni sull'intero database per capire ad esempio se supporta o meno certe caratteristiche dello standard SQL, esiste la classe DatabaseMetaData, che ha un'infinità di metodi (sembra abbia il pri-

mato della classe con più metodi di tutto Java). Il programma di esempio DatabaseMetaDataDemo mostra quasi tutte le informazioni che si possono ricavare da un database. Il codice è del tutto banale, il programma esegue una connessione al database ed esegue poi l'istruzione:

```
// Recupera metadati
DatabaseMetaData dbmd = conn.getMetaData();
```

per recuperare l'interfaccia DatabaseMetaData, dopodiché esegue quasi tutti i suoi metodi e ne visualizza i risultati sul terminale. Il lettore è invitato ad eseguire il programma sul proprio database; ecco il risultato del programma su un server Oracle (leggermente troncato per ragioni di spazio):

```
Data definition causes transaction commit: true
Data definition ignored in transaction: false
Does max row size include blobs: true
Catalogs:
Tables types:
    SYNONYM
    TABLE
    VIEW
Catalog preferred name:
Database name: Oracle
Database version: Oracle9i Enterprise Edition Release 9.2.0.1.0 - 64bit Production With the Partitioning,
OLAP and Oracle Data Mining options JServer Release 9.2.0.1.0 - Production
Database default trans. isolation: 2
Identifier quote string: "
Max column name length: 30
…
Max columns in 'select': 0
Max columns in a table: 1000
Max possible connections: 0
Max row size: 2000
Max statement length: 65535
Max number of open statements: 0
Max user name length: 30
System functions: USER
Numeric functions: ABS,ACOS,ASIN,ATAN,ATAN2,CEILING,COS,EXP,FLOOR,LOG,LOG10,MOD,
                   PI,POWER,ROUND,SIGN,SIN,SQRT,TAN,TRUNCATE
String functions: ASCII,CHAR,CONCAT,LCASE,LENGTH,LTRIM,REPLACE,RTRIM,SOUNDEX,SUBSTRING,UCASE
Time/Date functions: CURDATE,CURTIME,DAYOFMONTH,HOUR,MINUTE,MONTH,NOW,SECOND,YEAR
Procedure term: procedure
Schema term: schema
Schemas:
```

```
    MOKA
    NICO
    ORACLE
…
    SYSTEM
Search string escape: //
Not SQL 92 SQL keywords:
ACCESS, ADD, ALTER, AUDIT, CLUSTER, COLUMN, COMMENT, COMPRESS, CONNECT,
                    DATE, DROP, EXCLUSIVE, FILE, DENTIFIED, IMMEDIATE, INCREMENT, INDEX, INITIAL,
                    INTERSECT, LEVEL, LOCK, LONG, MAXEXTENTS, MINUS, MODE, NOAUDIT, NOCOMPRESS,
                    NOWAIT, NUMBER, OFFLINE, ONLINE, PCTFREE, PRIOR, all_PL_SQL_reserved_ words
Type info:
    INTERVALDS
    INTERVALYM
    TIMESTAMP WITH LOCAL TIME ZONE
    TIMESTAMP WITH TIME ZONE
    NUMBER
    NUMBER
    NUMBER
    LONG RAW
    RAW
    LONG
    CHAR
    NUMBER
    NUMBER
    NUMBER
    FLOAT
    REAL
    VARCHAR2
    DATE
    DATE
    TIMESTAMP
    STRUCT
    ARRAY
    BLOB
    CLOB
    REF
Database URL: jdbc:oracle:thin:@lunarossa.italy.sun.com:1521:sunone
Supports alter table with add column: true
Supports ANSI 92 Full: false
Supports Minimum SQL grammar: true
Supports Extended SQL grammar: true
…
```

Eseguendo invece il programma su un database DBAccess di nome mokadb si ottiene:

```
Data definition causes transaction commit: true
Data definition ignored in transaction: false
Does max row size include blobs: false
Catalogs:      D:\lavoro\attivita\Mokabyte\Manuale_Pratico_di_Java\2aEdizione\esempi\etc\mokadb
Tables types:
    SYNONYM
    SYSTEM TABLE
    TABLE
    VIEW
Catalog preferred name: DATABASE
Database name: ACCESS
Database version: 04.00.0000
Database default trans. isolation: 2
Identifier quote string: `
Max column name length: 64
Max columns in 'group by': 10
Max columns in 'order by': 10
Max columns in 'select': 255
Max columns in a table: 255
Max possible connections: 64
Max row size: 4052
Max statement length: 65000
Max number of open statements: 0
Max user name length: 0
System functions:
Numeric functions: ABS,ATAN,CEILING,COS,EXP,FLOOR,LOG,MOD,POWER,RAND,ROUND,SIGN,SIN,SQRT,TAN
String functions: ASCII,CHAR,CONCAT,LCASE,LEFT,LENGTH,LOCATE,LOCATE_2,LTRIM, RIGHT,RTRIM,SPACE, SUBSTRING,UCASE
Time/Date functions:  CURDATE,CURTIME,DAYNAME,DAYOFMONTH,DAYOFWEEK,DAYOFYEAR,
                      HOUR,MINUTE,MONTH,MONTHNAME,NOW,QUARTER,SECOND,WEEK,YEAR
Procedure term: QUERY
Schema term:
Schemas:
Mokaproblem: [Microsoft][ODBC Microsoft Access Driver]Optional feature not implemented
```

Il programma termina con una eccezione sul metodo DatabaseMetaData.getSchemas().

JDBC 1.1 core API: struttura del package java.sql

Questa sezione tratta con maggiore dettaglio alcune classi importanti del package java.sql, che appartenevano originariamente alla versione 1.1 di JDBC e che ora rientrano, insieme a

quelle di JDBC 1.2, nell'unico package java.sql.* distribuito con ogni versione di Java. Il package javax.sql.* contiene funzionalità avanzate di JDBC quali il Connection Pooling e le transazioni distribuite.

Connection

Di questa fondamentale classe si è già parlato e la si è già utilizzata in alcuni esempi. Due suoi metodi, normalmente poco utilizzati:

```
getTransactionIsolation()
setTransactionIsolation()
```

richiedono un approfondimento e una piccola digressione sulle transazioni.

È teoricamente dimostrabile, oltre che di per sé intuitivo, che per avere l'assoluta consistenza di una base dati le transazioni su di essa devono essere serializzate. Il grado più drastico di serializzazione consiste nell'eseguire una transazione, e mettere in attesa tutte le altre, finché questa non è terminata. Ciò è però inammissibile in ambienti dove non solo è possibile eseguire i programmi in modo concorrenziale, attraverso il time-sharing, ma soprattutto vi sono effettivamente le risorse (più di una CPU) per eseguire contemporaneamente più operazioni.

Occorreva quindi trovare un compromesso. La soluzione emersa per i database commerciali prevede l'interlacciamento e il controllo della esecuzione concorrente attraverso i *lock*, e la introduzione di livelli di transazione non puramente serializzabili.

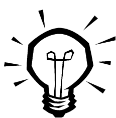
Schematizzando, i database usano due tipi di lock – scrittura (WL) e lettura (RL) – e un'algoritmo detto Two-Phase Locking per controllare l'esecuzione concorrente delle transazioni. Le regole di questo algoritmo sono tre, molto semplici:

1. Quando una transazione cerca di accedere a un certo record a scopo di lettura, il database alloca per essa un lock, detto lock di lettura RL. Analogamente se l'operazione è di modifica viene allocato un lock di scrittura, WL.

2. Il database scheduler mette in stato di WAIT, una transazione che implica l'allocazione di un lock su un record per il quale già esiste un lock allocato per un'altra transazione, conflittuale con il primo. Due lock non sono in conflitto solo se sono entrambi di sola lettura (RLs).

3. Una transazione non può rilasciare un lock e poi acquisirne un'altro, ovvero esistono, per una transazione, due fasi: la prima, durante la quale la transazione acquisisce i lock necessari per operare sui record, la seconda, durante la quale la transazione rilascia tutti i lock precedentemente acquisiti (generalmente questo accade quando si esegue l'operazione di COMMIT).

Un database che utilizza i RL/RW lock e il precedente algoritmo per serializzare le transazioni sulla base dati può garantire la consistenza delle informazioni.



Va ricordato che quasi tutti i database commerciali più diffusi hanno il locking a livello di riga di tabella. Ciò vuol dire che la risorsa minima che il database server riesce a controllare per accessi concorrenti è il record di tabella. Database meno potenti bloccano l'intera tabella limitando fortemente la possibilità di più transazioni di lavorare in modo parallelo e indipendente sulle stesse entità di database.

Pur permettendo l'interlacciamento e il parallelismo delle transazioni, la precedente implementazione, che prevede sempre la serializzazione delle transazioni, viene ritenuta eccessivamente penalizzante dal punto di vista delle performance. Non sempre d'altra parte è necessario che una transazione sia completamente isolata dalle altre. Per esempio, una transazione A di una certa applicazione sta aggiornando la colonna mail_ricevute della tabella articolisti del nostro database di esempio. Questa colonna indica il totale delle mail ricevute dall'articolista dai lettori di MokaByte. Una seconda transazione B di un'altra applicazione ha necessita di accedere a tutti i record della medesima tabella per estrarre un report relativo agli stessi articolisti. Secondo le regole del precedente algoritmo, la transazione B entra in WAIT finché la transazione B non ha terminato le operazioni di modifica perché i record sono in write lock. In realtà dal punto di vista della seconda applicazione il fatto che il campo mail_ricevute sia stato solo parzialmente scritto su tutti i record della transazione A o meno può non avere importanza, soprattutto se il report che deve creare non comprende l'informazione del campo mail_ricevute.

Per queste ragioni il comitato ANSI SQL99, adeguandosi oltretutto al fatto che molti produttori di database avevano già implementato qualcosa di simile, decise di introdurre quattro livelli di isolamento, di cui solo l'ultimo, SERIALIZABLE, corrisponde a quello teorico. I possibili livelli di isolamento della transazione sono riportati di seguito (in ordine di isolamento crescente; tra parentesi vi è la relativa costante da utilizzare in setTransactionIsolation()).

- *uncommitted read* (TRANSACTION_READ_UNCOMMITTED). Questo livello di isolamento permette di leggere anche i dati in transazione, quindi dati che stanno per essere modificati e non sono "integri".

- *committed read* (TRANSACTION_READ_COMMITTED). Lettura di soli dati non in transazione. Se un'altra sessione sta lavorando sui dati ai quali cerchiamo di accedere, la lettura viene bloccata.

- *repeatable read* (TRANSACTION_REPEATABLE_READ). Questo livello di isolamento vi garantisce non solo che i dati che leggete durante la vostra transazione siano integri ma anche che, se all'interno della stessa transazione leggete più volte gli stessi dati (ad esempio rifate la stessa query), riotterrete gli stessi risultati. Il livello di isolamento precedente non garantisce questa possibilità.

- *serializable* (TRANSACTION_SERIALIZABLE). È il livello di isolamento più alto per una transazione utente. Oltre a ovviare ai problemi precedenti, conosciuti come *dirty read* e *non-*

repeteable read, permette di affrontare anche il problema dei *phantom read*, che si verifica quando un'altra transazione inserisce ad esempio una nuova riga che soddisfa a una clausola di WHERE di una lettura all'interno della transazione. In tal caso i valori letti in precedenza non risultanoi modificati ma capiterà di vedere dati nuovi (*phantom*) che prima non comparivano nel record set. Questo livello di isolamento corrisponde a quello che prevede la pura e semplice serializzazione e garantisce l'assoluta consistenza del database.

Perché non impostare fin dall'inizio, o prima di qualsiasi transazione, il massimo livello in modo da evitare in anticipo ogni problema? La risposta è semplice: l'aumento del livello di isolamento di una o più sessioni utenti limita il grado di parallelismo del database, non permettendo al server di eseguire le transazioni in modo concorrente. Ne risulta quindi un abbassamento delle performance. Il più delle volte è il sufficiente il livello di isolamento di default, che può essere reperito con il metodo DatabaseMetaData.getDefaultTransactionIsolation(). Nella sezione "Metadati" abbiamo mostrato l'output del programma DatabaseMetaDataDemo che visualizza per MDB di Microsoft e Oracle il livello di default di isolamento della transazione (COMMITTED READ) per i due database.

Nel piccolo esempio che segue viene mostrato il problema della non-ripetibilità delle letture quando si è nel livello READ COMMITTED. Il programma di demo che utilizziamo, QueryUpdateShell.java, è una fusione dei due programmi già presentati UpdateShell.java e QueryShell.java e permette all'utente di fare sia query che aggiornamenti, utilizzando lo stessa mini-shell. Il programma, oltre ai parametri usuali (parametri di connessione, username, password), accetta un argomento per indicare il livello di isolamento della transazione. Per il nostro esperimento occorre eseguire due istanze del programma da due shell DOS o UNIX nel modo seguente (utilizzando quindi lo script di start):

```
D:\Manuale_Pratico_di_Java\esempi> runQueryUpdateShell read_commited
```

A questo punto dalla shell 1 eseguire i seguenti comandi:

```
<QueryShell>$ update
<UpdateShell>$ create table tab0 (num int);
Aggiornato/i 0 records.

<UpdateShell>$ insert into tab0 values(1);
Aggiornato/i 1 records.

<UpdateShell>$ insert into tab0 values(2);
Aggiornato/i 1 records.

<UpdateShell>$ insert into tab0 values(3);
Aggiornato/i 1 records.
```

Il primo comando serve al programma per dire che si desidera fare aggiornamenti e non query.

Dalla shell 2 invece:

```
<QueryShell>$ select * from tab0;
Eseguo query!
[NUM]
```

La seconda transazione (seconda shell) vede la nuova tabella, ma non ancora i dati, perché dalla prima transazione non abbiamo ancora effettuato una commit. A rigore non avremmo neanche dovuto vedere la tabella, ma Oracle fa un implicito commit per le operazioni di Data Definition (`Create` ...). A questo punto dalla shell 1 digitiamo:

```
<UpdateShell>$ commit

Dall shell 2 invece:

<QueryShell>$ select * from tab0;
Eseguo query!
[NUM]
1
2
3
```

Vediamo i dati perché la transazione 1 ha effettuato una commit, come richiesto dal livello di isolamento `READ COMMITTED`.

Il problema sorge con questo livello di isolamento quando ad esempio la transazione 2 legge due volte nella stessa transazione i dati nella tabella `tab0` e si ritrova come result set diversi. Ciò può portare a inconsistenze.

Ripetiamo ora l'esperimento avviando così il programma dalle due shell (il lettore ricordi che la tabella è già stata creata nell'esempio precedente):

```
D:\Manuale_Pratico_di_Java\esempi> runQueryUpdateShell read_commited
(shell 1)

D:\Manuale_Pratico_di_Java\esempi> runQueryUpdateShell serializable
(shell 2)
```

Dalla shell 1 eseguiamo i seguenti aggiornamenti:

```
<QueryShell>$ update
<UpdateShell>$ insert into tab0 values(99);
Aggiornato/i 1 records.

<UpdateShell>$ insert into tab0 values(100);
```

```
Aggiornato/i 1 records.

<QueryShell>$ commit
<UpdateShell>$ query
<QueryShell>$ select * from tab0;
Eseguo query!
[NUM]
1
2
3
99
100
```

Dalla seconda shell invece:

```
<QueryShell>$ select * from tab0;
Eseguo query!
[A]
1
2
3
```

Questa volta alla seconda transazione (programma eseguito nella seconda shell) non sono visibili le modifiche apportate dalla prima transazione, nemmeno dopo il COMMIT. Sarebbe bastato avviare la seconda istanza del programma nel modo seguente:

```
D:\Manuale_Pratico_di_Java\esempi> runQueryUpdateShell repeatable_read
  (shell 2)
```

Oracle Server però, su cui è stato eseguito il programma, supporta solo i livelli di isolamento della transazione READ COMMITTED e SERIALIZABLE.

Viene di seguito riportato un estratto del codice del programma per comodità del lettore:

```
...
        // Imposta livello di isolamento della connessione richiesto
        // dall'utente e commit manuale
        conn.setAutoCommit(false);
        conn.setTransactionIsolation(getIsolationFromString(args[1]));

        // Creazione dello statement, l'oggetto che utilizzo per
        // spedire i comandi SQL al database
        Statement stmt = conn.createStatement();
...
```

```
String[] prompt = new String[2];
prompt[0] ="<QueryShell>$ ";
prompt[1] = "<UpdateShell>$ ";

ResultSet rs = null;
ResultSetMetaData rsmd = null;

while(true){
  out.print(prompt[statement_type]);
  out.flush();

  // prende SQL
  while((sql = in.readLine()).equals("")) {
    out.print(prompt[statement_type]);
    out.flush();
  }

  if(sql.equalsIgnoreCase("exit") || sql.equalsIgnoreCase("quit")
    || sql.equalsIgnoreCase("q") || sql.equalsIgnoreCase("x"))
    break;

  // l'utente vuole ora fare delle query
  if(sql.equalsIgnoreCase("query") || sql.equals("Q")){
    statement_type = 0;
    continue;
  }

  // l'utente vuole ora fare degli update
  if(sql.equalsIgnoreCase("update") || sql.equals("U"))  {
    statement_type = 1;
    continue;
  }

  // richiesto commit o rollback ?
  if(sql.equalsIgnoreCase("commit") || sql.equals("C")){
    conn.commit();
    conn.setAutoCommit(false);
    continue;
  }

  // richiesto commit o rollback ?
  if(sql.equalsIgnoreCase("rollback") || sql.equals("R")){
    conn.rollback();
```

```
    conn.setAutoCommit(false);
    continue;
}

// rimuove il ';' finale, per evitare errore da parte
// del driver JDBC Oracle.
// Usando il driver JDBC Informix questo problema non c'è.
if(sql.charAt(sql.length() - 1) == ';')
    sql = sql.substring(0, sql.length() - 1);

// esegue una query
if(statement_type == 0){
    try{
        System.out.println("Eseguo query!");
        rs = stmt.executeQuery(sql);
        rsmd = rs.getMetaData();
        int cols = rsmd.getColumnCount();

        // Stampa i nomi delle colonne
        // Nota: le colonne hanno indice che parte da 1
        String name = null;
        for (int i = 1; i <= cols; ++i) {
            out.print("[");
            out.print(rsmd.getColumnLabel(i));
            out.print("]");
            //if(i != cols)
            // out.print("|");
        }
        out.print('\n');

        // stampa le righe
        while(rs.next()) {
            for (int i = 1; i <= cols; ++i) {
                out.println(rs.getString(i));
                if(i != cols)
                    out.print("|");
            }
        }
        out.print("\n");

        rs.close();
    }
    catch(SQLException sqle) {
```

```
            System.err.println("Mokaproblem: executeQuery:" + sqle.getMessage());
            conn.rollback();
            conn.setAutoCommit(false);
            continue;
         }
      }

      // Eseguo comando di aggiornamento
      if(statement_type == 1) {
         try{
            updret = stmt.executeUpdate(sql);
            out.println("Aggiornato/i " + updret + " records.\n");
            out.flush();
         } catch(SQLException sqle) {
            System.err.println("Mokaproblem: executeUpdate: "
               + sqle.getMessage());
            continue;
         }
      }

   } // while
```

Di seguito si affronterà la descrizione di alcuni metodi:

```
commit()
rollback()
rollback(Savepoint)
getAutoCommit()
setAutoCommit(boolean)
setSavePoint()
setSavePoint(String)
releaseSavePoint(Savepoint)
```

Questi metodi si utilizzano per avviare e controllare una transazione in modo esplicito e programmatico, senza cioè inviare al database le istruzioni SQL classiche BEGIN WORK, COMMIT WORK, ROLLBACK. I savepoint si riferiscono alla caratteristica di alcuni database (come ad esempio Oracle, ma non Informix) di effettuare il rollback di una transazione fino a un punto (savepoint) in cui il programma ritiene che le modifiche apportate siano comunque consistenti.

Normalmente tutti i comandi SQL eseguiti da un'applicazione sono svolti in modalità autocommit. Ciò vuol dire che per ogni singolo comando SQL viene aperta e chiusa una transazione, in modo trasparente per il client. Nel caso invece occorra rendere atomica una serie di operazioni SQL che hanno senso se eseguite insieme, bisogna disabilitare la modalità autocommit con l'istruzione:

```
setAutoCommit(false); // disabilita autocommit e *implicitamente* avvia transazione
```

Questa istruzione automaticamente avvia una transazione – non occorre insomma un'ulteriore istruzione del tipo stmt.execute("BEGIN WORK") – che sarà terminata con una delle due istruzioni commit() o rollback().

Si vedranno in seguito esempi di utilizzo delle transazioni e dei metodi mostrati. Potete ovviamente utilizzare questi metodi solo se le transazioni per il database che state utilizzando sono supportate: per saperlo a priori utilizzate il metodo Database-MetaData.supportsTransactions().

```
prepareStatement()
prepareCall()
```

Si utilizzano questi due metodi rispettivamente per creare uno statement preparato e una chiamata a una stored procedure. Per statement preparato si intende uno statement SQL parametrizzato, dove cioè alcune parti, tipicamente i valori delle clausole di WHERE, sono variabili, e possono essere valorizzate successivamente, al momento della esecuzione dello statement.

Oltre al fatto che il codice appare molto più consistente, il vero vantaggio di utilizzare questo tipo di statement, quando ha senso, è prestazionale. Normalmente i database server hanno un modulo, detto SQL Parser, che si occupa della compilazione dello statement (in linguaggio interno del database), della sua validazione (per esempio, controlla se le tabelle o colonne richieste dalla query esistono e l'utente ha i permessi per accedervi) e della definizione di execution plan per lo statement (per capire ad esempio se la query va eseguita utilizzando gli indici o attraverso un full table scan, il tipo di algoritmo di ordinamento dei risultati). Utilizzando statement preparati queste operazioni da parte del DB server vengono fatte solo una volta e per ogni successiva invocazione di pstmt.executeQuery() si ha solo l'esecuzione dello statement.

In modo simile si esegue la chiamata a una stored procedure nel database. Nei database si distingue tra stored procedure e stored function. Quest'ultime si differenziano per il solo fatto che viene restituito un risultato. Tradizionalmente le stored procedure/function vengono scritte con un linguaggio di script interno, che per esempio in Oracle Server si chiama PL/SQL. Con l'integrazione di Java nei database server oggi è possibile scrivere (sempre ovviamente se il database lo supporta) anche stored procedure in Java, utilizzando SQLJ. Una stored procedure Java è una normale classe Java JDBC che per lo più utilizza uno speciale driver JDBC interno e con uno o più metodi statici che costituiscono la "procedure" o "function" SQL. La sintassi ad ogni modo per accedere a questo tipo di stored procedure è la stessa:

```
...
CallableStatement cs = conn.prepareCall("{CALL PROC controlla_utente(?)}");
CallableStatement cs = conn.prepareCall("{? = CALL FUNC conta_articoli_tipo(?,?)}");
…
```

Statement

Lo statement è l'interfaccia che le applicazioni utilizzano per spedire comandi SQL al database. È un concetto generale valido non solo per Java, perché esiste in tutte le API C di basso livello di tutti i database.

```
execute()
executeQuery()
executeUpdate()
```

Questi metodi, e in particolare gli ultimi due, sono da utilizzare per ogni operazione sul database. Il primo è un metodo generico, utilizzabile per istruzioni SQL tipo DDL e DML; il secondo è specializzato per le query o in generale per operazioni che ritornano un result set, una collezione di dati; executeUpdate(), si utilizza invece per tutte le operazioni che non ritornino un result set ma al massimo un update count (come nel caso di cancellazioni, inserimenti e aggiornamenti) ma anche per istruzioni CREATE TABLE..., CREATE TRIGGER…, etc.

Il primo metodo, che viene usato di rado, è interessante anche perché permette di eseguire in una sola volta istruzioni multiple separate dal punto e virgola come:

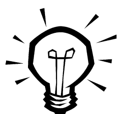
```
…
String sql = "SELECT * FROM ARTICOLI;" +
            "DELETE FROM articoli WHERE cognome = 'Venditti';"
            "UPDATE articoli SET nome = 'Giovanni' WHERE nome = 'Giovannino' AND cognome = 'Puliti';"
            "SELECT count(*) FROM ARTICOLI WHERE cognome = 'Puliti';"
stmt.execute(sql);
…
```

Per recuperare i vari risultati si utilizzano i metodi getMoreResults(), getUpdateCount(), getResultSet(). Il metodo Statement.getMoreResults() restituisce true se esistono ancora risultati da leggere; nel caso ve ne siano si chiama Statement.getUpdateCount(). Se questa chiamata vi restituisce –1 allora vuol dire che il risultato corrente è un result set a cui potete accedere con getResultSet():

```
…
restyp = stmt.execute(sql);
if(restyp){
  // è un result set, leggi risultati
}else{
  // è un update count
}
boolean restyp = false;
for(;;){
  restyp = stmt.getMoreResults();
if(restyp){
  // è un result set, leggi risultati
} else {
  if(stmt.getUpdateCount() == -1) {
    // fine risultati
    break;
```

```
    } else {
        // è un update count
    }
}
}
…
```

Nel caso della query di esempio sopra riportata, l'istruzione SQL restituirà nell'ordine: un result set (SELECT * FROM ARTICOLI), un update count (DELETE…), un altro update count (UPDATE…) e ancora un result set di una riga (SELECT…). Grazie a execute() si può implementare un editor SQL in Java capace di eseguire blocchi di istruzioni scritti dall'utente e di visualizzarne il contenuto in modo formattato, il che evita di dover fare il parsing del testo per distinguere le operazioni di query, per le quali si utilizza executeQuery, da quelle di update.



Per istruzioni SQL del tipo SELECT * FROM ARTICOLI INTO TEMP temp1 è preferibile utilizzare executeUpdate() invece di executeQuery() perché la clausola INTO TEMP crea una tabella temporanea, quindi altera, anche se in modo non esplicito, lo schema del database, cosa che una normale select non fa. Il valore restituito, il numero delle righe inserite nella tabella temporanea, è appunto un update count.

PreparedStatement

Come già anticipato nella precedente sezione se si prevede di utilizzare spesso la stessa istruzione SQL, conviene "prepararla", ovvero fare in modo che il DB server possa precompilarla per poi rieseguirla con i soli parametri variabili. Ad esempio, query come la seguente:

```
SELECT * FROM articoli WHERE nome = 'Nicola' AND cognome = 'Venditti'
SELECT * FROM articoli WHERE nome = 'Giovanni' AND cognome = 'Puliti'
```

possono essere preparate con la sintassi seguente:

```
SELECT * FROM articoli WHERE nome = ? AND cognome = ?
```

In codice diventa:

```
…
ResultSet rs = null;
String sql = "SELECT * FROM articoli WHERE nome = ? AND cognome = ?";
PreparedStatement pstmt = conn.prepareStatement(sql);
```

```
// recupera articoli di Nicola
pstmt.setString(1,'Nicola');
pstmt.setString(1,'Venditti');
rs = pstmt.executeQuery();
...

// recupera articoli di Giovanni
pstmt.setString(1,'Giovanni');
pstmt.setString(1,'Puliti');
rs = pstmt.executeQuery();
...
```

Dove possibile, è meglio utilizzare sempre i prepared statement invece dei normali statement. Oltre a semplificare e razionalizzare il codice, migliorano le prestazioni del database.

CallableStatement

Del CallableStatement abbiamo già parlato; vediamo ora un esempio di chiamata a stored procedure PL/SQL di Oracle. Prima di tutto creiamo la funzione PL/SQL con SQL*PLUS:

```
<oracle@mokamachine> ~$ sqlplus moka/corretto
SQL> CREATE OR REPLACE FUNCTION conta_lettori(pub INTEGER)
RETURN INTEGER IS cnt INTEGER;
BEGIN
  IF pub = 0 THEN
      SELECT  count(iduser) INTO cnt
      FROM    lettori
      WHERE   non_pubblico = 'S';
  ELSE
      SELECT  count(iduser) INTO cnt
      FROM    lettori
      WHERE   non_pubblico = 'N';
  END IF;
  RETURN cnt;
END;
/
SQL> SHOW ERRORS
SQL> QUIT
```

A questo punto si può eseguire il seguente codice per accedere alla funzione PL/SQL appena creata:

```
…
CallableStatement cstmt = conn.prepareCall("{? = CALL balance(?)}");
```

```
cstmt.registerOutParameter(1, Types.INTEGER);
cstmt.setInt(2, 1);
cstmt.executeUpdate();
int cnt = cstmt.getInt(1);
…
```

La stored procedure appena creata è piuttosto banale. Normalmente vi si ricorre per operazioni più complesse, oppure per nascondere alcuni dettagli dello schema fisico del database. Per esempio, nel nostro caso, i programmi client chiamano la procedura contami() per contare i lettori di MokaByte, senza sapere quale sia effettivamente la tabella che contiene i dati, né la sua struttura. Altro vantaggio è che, essendo lo schema fisico non direttamente visibile, il DBA è libero di riorganizzare i dati come crede, per esempio distribuendo le informazioni dei lettori in due tabelle, lettori_pubblici e lettori_privati, per poi riscrivere la stored procedure di conseguenza: per i programmi client non cambia una virgola.

ResultSet

Il result set, altrove chiamato anche Record Set, è un oggetto dinamicamente generato dallo statement per la manipolazione dei dati restituiti dalle query SQL.

```
getXXX()
```

Questa famiglia di metodi, a cui appartengono ad esempio getString() e getInt(), si utilizza per recuperare i dati presenti nel result set. Esiste un metodo getXXX() per ogni tipo di dato SQL.

Per la maggior parte dei campi è possibile utilizzare getString(), per farsi restituire in formato di stringa il valore del risultato. Vi sono alcuni metodi per il recupero dei dati come stream Java che meritano un esempio. Ad alcuni tipi di dati di database è possibile (o è necessario) accedere utilizzando gli stream Java, InputStream e OutputStream.

L'esempio di codice che segue mostra come estrarre una immagine GIF da una colonna di database e salvarla su file:

```
…
ResultSet rset
= stmt.executeQuery ("SELECT foto FROM autori WHERE nome = 'NICOLA' and cognome='VENDITTI'");

// abbiamo una sola riga
if (rset.next()){
  // Accede alla foto attraverso uno stream binario
  InputStream foto_stream = rset.getBinaryStream (1);
  try {
    FileOutputStream file = null;
    file = new FileOutputStream ("Foto_Venditti.gif");
    int data;
    while ((data = foto_stream.read()) != -1)
```

```
            file.write(data);
        }  catch (Exception e)  {
            System.out.println(e);
        }
        finally {
            if file != null
    file.close();
        }
    }
```

La tabella autori è quella del database di esempio mokadb, e la colonna foto ha come tipo ORACLE il LONG RAW, che è appunto tipo di dato binario.

```
wasNull()
```

Supponiamo che per l'articolista "Mario Rossi" non sia pervenuta l'informazione sulla sua età. L'amministratore del DB decide, logicamente, di inserire un valore null nel campo della relativa riga della tabella ARTICOLISTI. Il valore null in un database non vuol dire 0 o stringa nulla ma qualcosa come "campo non valorizzato" o "valore non applicabile".

Cosa succede quando un'applicazione contenente del codice come quello appena mostrato legge l'età di "Mario Rossi"? In questi casi occorre chiamare ResultSet.wasNull(), subito dopo la lettura della colonna, in questo caso getInt("eta"), per capire se contiene un null. In caso negativo viene mostrato il valore restituito dalla lettura.

JDBC 2.x core API

Le versioni 2.x del JDBC introducono alcune fondamentali aggiunte che si integrano perfettamente nell'impianto originario. Il fatto che nulla del design originario sia cambiato indica quanto siano state oculate le scelte progettuali alla base di JDBC: non per tutte le librerie Java è stato così. Le aggiunte introdotte in JDBC 2 possono essere raggruppate sotto due categorie:

Nuove funzionalità

- result set scrollabili;
- operazioni DML/DDL programmatiche;
- batch update;
- aggiunte minori per l'aumento delle performance, per il supporto delle time zone, etc.

Supporto per i nuovi tipi di dati

- supporto per i nuovi tipi astratti SQL3;

- supporto per l'archiviazione diretta degli oggetti nel database.

Cominciamo dalle nuove funzionalità.

Nuove funzionalità di JDBC

Result set scrollabili

Chi è abituato a lavorare con Access, per fare un esempio, trova naturale eseguire una query, recuperare il record set (il result set secondo la terminologia di ODBC) e scandirlo in avanti o all'indietro, posizionare il cursore dove si vuole o avere il count degli elementi. Con JDBC 1.1 questo non è possibile: una volta recuperato il result set lo si poteva scandire solo in avanti e non era possibile avere il numero delle righe restituite in anticipo. Perché con Access è possibile fare una cosa in apparenza così semplice che con JDBC sembra impossibile?

La limitazione sta in parte in JDBC e in parte nel meccanismo con cui il database restituisce i dati ai client. Come accennato altrove il result set, dal punto di vista del database è un cursore posizionato all'inizio della tabella fittizia che contiene i risultati che soddisfano la query. Una volta spostato in avanti il cursore non è possibile ritornare indietro normalmente: normalmente vuol appunto dire che il tipo di cursore allocato per l'operazione di lettura dei risultati è unidirezionale, questo perché è meno impegnativo dal punto di vista delle risorse.

Quasi tutti i database, compresi Oracle e Informix utilizzano questo metodo. In alcuni casi, per venire incontro ai programmatori e fare in modo che possano utilizzare comunque questa possibilità, i produttori di database hanno fornito dei driver ODBC con un'opzione impostabile dall'amministratore della sorgente di dati ODBC per l'uso dei record set scrollabili.

Con JDBC 2 il programmatore Java ha la possibilità di scegliere da programmazione l'uso di questa opzione, specificandolo nel normale processo di creazione dello statement. Ad esempio:

```
...
Connection conn
= DriverManager.getConnection("jdbc:informix-sqli://db.mokabyte.it:1526/mokadb:INFORMIXSERVER
=ol_mokabyte", "informix", "mokapass");

Statement stmt = conn.createStatement(ResultSet.SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
...
```

In questo caso creo un cursore insensibile ai cambiamenti apportati da altri sullo stesso result set, e di sola lettura. Il supporto completo per lo scrollable result set è così articolato:

- L'oggetto ResultSet è stato dotato di metodi opportuni per il posizionamento casuale del cursore oltre che di attributi da utilizzare come costanti per la definizione del tipo di result set che si vuole.

- All'oggetto Connection sono stati aggiunti degli overload per il metodo createStatement() per permettere al programmatore di poter definire il tipo di cursore desiderato.

- All'interfaccia DatabaseMetaData sono stati aggiunti metodi perché l'applicazione possa accertarsi a priori se il database supporta questa funzionalità.

Vediamo un esempio abbastanza completo di utilizzo dei nuovi metodi per la gestione dei result set scrollabili. Il programma ScrollableResultSetDemo.java permette all'utente di effettuare una query e di navigare il result set.

```
// Legge da stdin query da eseguire
//
out.print("Query SQL(quit per uscire): ");
out.flush();

while((sql = in.readLine()).equals(""))
  out.print("\nQuery SQL(quit per uscire): ");;

if(sql.equals("exit") || sql.equals("quit"))
    System.exit(0);

// rimuove il ';' finale, per evitare errore da parte
// del driver JDBC Oracle.
// Usando il driver JDBC Informix questo problema non c'è.
if(sql.charAt(sql.length() - 1) == ';')
  sql = sql.substring(0, sql.length() - 1);

DriverManager.println(">> Esecuzione query <" + sql + ">");

// Eseguo la query o un altro comando SQL attraverso lo statement
// e recupero i risultati attraverso l'interfaccia ResultSet
//
try {
  rs = stmt.executeQuery(sql);
  rsmd = rs.getMetaData();
} catch(SQLException sqle) {
  System.err.println("Mokaproblem: " + sqle.getMessage());
  System.exit(0);
}

// Menu utente interattivo
//
out.println("Query eseguita. Inserire ora i comandi per lavorare sul resultset");
out.println("Digitare 'help' per aiuto in linea.");
out.println();
out.flush();
```

```
String cmd = null;

StringTokenizer st = null;
while(true){

  // Legge comando
  //
  out.print("<<ResultSet>> ");
  out.flush();

  while((cmd = in.readLine()).equals(""))
    out.print("\n<<ResultSet>> ");

  if(cmd.equalsIgnoreCase("HELP")) {
    out.println("\t  !!! HELP !!! ");
    out.println("\thelp: questo menu");
    out.println("\tquit/exit: termina programma");
    out.println("\tfirst: si posiziona sulla prima riga");
    out.println("\tlast: si posiziona sull'ultima riga");
    out.println("\tabsolute <n>: si posiziona alla riga <n>");
    out.println("\trelative [-]<n>: si posiziona [-]<n> relativamente alla posizione corrente");
    out.println("\tprint: stampa riga corrente");
    out.println("\tcount: stampa numero delle rige di questo result set");
    out.flush();
    continue;
  } else if(cmd.equalsIgnoreCase("QUIT") || cmd.equalsIgnoreCase("EXIT"))          {
    break;
  } else if(cmd.equalsIgnoreCase("first")) {
    rs.first();
  } else if(cmd.equalsIgnoreCase("last")) {
    rs.last();
  } else if(cmd.equalsIgnoreCase("beforeFirst")){
    rs.beforeFirst();
  } else if(cmd.equalsIgnoreCase("afterLast"))  {
    rs.afterLast();
  } else if(cmd.startsWith("abs")) {
    st = new StringTokenizer(cmd);
    st.nextToken(); // scarta primo token
    int n;
    try{
      n = Integer.parseInt(st.nextToken());
    }catch(Exception e){ break; };
      rs.absolute(n);
```

```
} else if(cmd.startsWith("rel")){
  st = new StringTokenizer(cmd);
  st.nextToken(); // scarta primo token
  int n;
  try{
    n = Integer.parseInt(st.nextToken());
  }catch(Exception e){ break; };

  rs.relative(n);
} else if(cmd.equalsIgnoreCase("print")) {
  int cols = rsmd.getColumnCount();

  // Stampa i nomi delle colonne
  // Nota: le colonne hanno indice che parte da 1
  String name = null;
  try {
    for (int i = 1; i <= cols; ++i) {
      out.print("[");
      out.print(rsmd.getColumnLabel(i));
      out.print("]");
    }
    out.print('\n');

    // stampa le righe
    rs.next();
    for (int i = 1; i <= cols; ++i) {
      out.print(rs.getString(i));
      if(i != cols)
        out.print("|");
    }
  }catch(SQLException sqle){
    System.err.println("Mokaproblem: " + sqle.getMessage());
  }
  out.print("\n\n");
} else if(cmd.equalsIgnoreCase("count")) {
  int n = 0;
  rs.beforeFirst();
  while(rs.next())++n;
  out.println("\nNumero record result set: " + n);
} else {
  out.println("\nComando non riconosciuto!");
  out.flush();
}
```

```
}
...
```

Il programma è abbastanza semplice: nella prima parte si prende la query da eseguire dall'utente e si interroga il database, nella seconda invece il programma interagisce con l'utente e agisce in base al comando di navigazione inserito.

Cancellazione, inserimento e update programmatici

La maniera tradizionale di aggiornare una riga di una tabella è quella di eseguire un comando SQL di UPDATE. Ad esempio, la seguente istruzione:

```
...
stmt.executeUpdate("UPDATE servizi SET stato = 'SOSPESO' WHERE data_avvio is NULL");
...
```

aggiorna il prezzo di un prodotto in catalogo; allo stesso modo si inserisce un nuovo prodotto o se ne cancella uno.

JDBC 2 introduce un modo alternativo di eseguire le istruzione di tipo DML come quelle viste. Per le operazioni di aggiornamento dei dati, JDBC mette ora a disposizione le istruzioni updateXXX(), che agiscono sui campi della riga corrente. I passi che il programma segue per aggiornare i dati sono i seguenti:

1. Connessione e creazione del result set (*updatables*).

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  // JDBC / ODBC
Connection conn = DriverManager.getConnection("jdbc:odbc:mokadb");
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
  ResultSet srs = stmt.executeQuery("SELECT * FROM articoli");
```

2. Posizionamento sulla riga che si vuole aggiornare e aggiornamento dei campi che interessano.

```
srs.first();
// aggiorna il titolo del primo articolo di MokaByte
srs.updateString("TITOLO", "Primo articolo di MokaByte!");
```

3. Aggiornamento della riga.

```
srs.updateRow();  // Conferma l'update
```

4. Chiusura della connessione.

```
srs.close();
```

```
stmt.close();
conn.close();
```

Nel caso si volessero annullare tutte le operazioni di aggiornamento fatte, prima di updateRow(), è sufficiente chiamare ResultSet.cancelRowUpdates(), e saranno ripristinati i valori dei campi al valore originario.

Nel caso di inserimento di una nuova riga si procede invece come segue:

1. Connessione e creazione del result set (*updatables*).

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
// via ODBC
Connection conn = DriverManager.getConnection("jdbc:odbc:mokadb");
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
  ResultSet srs = stmt.executeQuery("SELECT * FROM articoli");
```

2. Ci si posiziona sulla "riga di inserimento" – una riga fittizia introdotta per rendere possibile l'operazione di inserimento – con il nuovo metodo ResultSet.moveToInsertRow(), si usano i metodi updateXXX() per definire i campi della nuova riga e, infine, si ritorna alla precedente riga del result set.

```
srs.moveToInsertRow();
// imposta il titolo
srs.update("TITOLO", "Primo articolo di MokaByte!");
// id autore (nella tabella autori)
srs.update("IDAUTORE", 1200);
// data articolo
srs.update("DATA", "10/10/2000");
...
// inserisce la riga nella tabella ARTICOLI
srs.insertRow();
// ritorna alla precedente riga del result set
srs.moveToCurrentRow();
```

3. Chiusura della connessione

```
srs.close();
stmt.close();
conn.close();
```

L'istruzione srs.moveToCurrentRow(), permette di ritornare alla riga corrente del result set prima della chiamata a moveToInsertRow().

Ancora più semplice dell'inserimento di una riga è la sua cancellazione. Le istruzioni sono le seguenti:

```
// ci si posizione sulla riga contenente la riga da cancellare
srs.absolute(5):
// si cancella la riga dalla tabella
srs.deleteRow();
```

Possono sorgere delle inconsistenze nel result set corrente quando si cancellano o si inseriscono delle righe. Per esempio, nel caso di cancellazione di una riga, essa non dovrebbe più comparire nel result set. Ma alcuni driver JDBC introducono in luogo della riga cancellata una riga *blank* come rimpiazzo.

In generale, per fronteggiare problemi come questi occorre interrogare i metodi ownUpdatesAreVisible(), ownDeletesAreVisible() e ownInsertsAreVisible() dell'interfaccia DatabaseMetaData, che danno informazioni sul supporto fornito dal driver JDBC per le funzionalità da utilizzare.

Nel caso specifico è comunque possibile chiudere e riaprire il result set (attraverso una nuova query) per evitare qualsiasi problema di inconsistenza dei dati.

Batch update

Questa nuova possibilità, introdotta più per ragioni di performance che come nuova funzionalità applicativa, permette di spedire al database una serie di aggiornamenti in blocco (ovvero in *batch*) piuttosto che uno dopo l'altro: si da così al database la possibilità di introdurre delle ottimizzazioni nella sequenza di modifiche che deve apportare.

Anche in questo caso sono state modificate alcune classi per poter supportare questa aggiunta: le classi Statement, PreparedStatement, and CallableStatement hanno dei metodi in più, vale a dire addBatch(), clearBatch(), executeBatch(); alla classe DatabaseMetaData è stato aggiunto il metodo supportsBatchUpdates(); infine è stata introdotta una nuova eccezione, ovvero BatchUpdateException, lanciata nel caso di anomalie nel processo di batch update.

Ecco un esempio di codice che esegue una serie di operazioni DML in batch per introdurre tre nuovi articoli di un nuovo articolista di MokaByte.

```
…
try {
// apre una transazione
conn.setAutoCommit(false);

Statement stmt = conn.createStatement();
// inserisce dati anagrafici dell'articolista
stmt.addBatch("INSERT INTO AUTORI " + "VALUES(1001, 'Mario', 'Rossi', 'marior@mokabyte.it')");
// inserisce primo articolo
stmt.addBatch("INSERT INTO ARTICOLI " + "VALUES('Enterprise JavaBeans Parte 1', 1001, '10/06/2000')");
```

```
// inserisce primo articolo
stmt.addBatch("INSERT INTO ARTICOLI " + "VALUES('Enterprise JavaBeans Parte 2', 1001, '10/07/2000')");
// inserisce primo articolo
stmt.addBatch("INSERT INTO ARTICOLI " + "VALUES('EJB e J2EE', 1001, '10/08/2000')");

// esegue batch
int [] updateCounts = stmt.executeBatch();
// chiude la transazione
conn.commit();
// default auto-commit
conn.setAutoCommit(true);

} catch(BatchUpdateException be) {
  System.err.println("---- BatchUpdateException ----"); System.err.println("SQLState: "
                              + be.getSQLState()); System.err.println("Messaggio err.: " + be.getMessage());
  System.err.println("Codice err.: " + be.getErrorCode()); System.err.print("Updates: ");
  int [] updateCounts = be.getUpdateCounts();
  for (int i = 0; i < updateCounts.length; i++) {
    System.err.print(updateCounts[i] + " ");
  }
} catch(Exception e) {
  System.err.println("--- EXCEPTION! ---");
  System.err.println(e.getMessage());
}
…
```

Anche per questa funzionalità è bene accertarsi che sia effettivamente supportata dal driver attraverso il metodo `DatabaseMetaData.supportsBatchUpdates()`.

Supporto per i tipi di dati avanzati

Tipi di dati SQL3 in Java

Per nuovi tipi di dati SQL3 si intendono i nuovi tipi di dato SQL introdotti dallo standard SQL3. SQL ha attraversato una serie di successive standardizzazioni: ISO/ANSI, SQL86, SQL89, SQL92 e ora SQL3. Con SQL2 e SQL3 sono stati introdotti i seguenti nuovi tipi di dati non elementari o complessi:

- `CLOB` (Character Large OBject). È un cosiddetto *smart blob space* perché, a differenza dei precedenti, forme di large object permettono un accesso casuale al dato stesso. Si utilizza per l'immagazzinamento di lunghe stringhe di caratteri (diciamo qualche kylobyte) come ad esempio descrizioni estese.

- BLOB (Binary Large OBject). Si utilizza per immagazzinare dati binari molto lunghi come immagini o documenti in formato binario.

- ARRAY. Permette di utilizzare un array come valore di una colonna (SQL3).

- ROW TYPE. Definisce il tipo di dato riga. È possibile ad esempio che una colonna di database sia strutturata, contenendo a sua volta una riga all'interno e violando così la prima forma normale del modello relazionale (come nel caso di un array).

- REF. Si tratta del tipo di dato "riferimento"; serve cioè per definire riferimenti a dati (SQL3).

Per ognuno di questi nuovi tipi di dato SQL, Java ha un oggetto o un'interfaccia che ne permette l'utilizzo nei programmi. Precisamente:

- un oggetto Clob corrisponde a un CLOB SQL;

- un oggetto Blob corrisponde a un BLOB SQL;

- un oggetto Array corrisponde a un ARRAY SQL;

- un oggetto Struct corrisponde a un ROW TYPE;

- un oggetto Ref corrisponde a un REF SQL.

Esiste ancora un altro tipo di dato SQL non citato in precedenza: il DISTINCT TYPE. In molti database è presente già da molto tempo. Non è indispensabile e normalmente lo si utilizza solo per affinare logicamente il modello della propria base dati. Un distinct type è, grosso modo, una ridefinizione di un tipo di dato primitivo esistente in un range di valori diverso dal default. Ad esempio:

```
CREATE DISTINCT TYPE CODART_T AS CHAR(8);
```

crea il nuovo tipo di dati COD_ARTICOLO che è un CHAR di esattamente 8 caratteri. Non esistendo il tipo CHAR(8) è indubbiamente una comodità poterlo definire prima di disegnare il modello del database. In tal modo infatti posso scrivere

```
— tabella articoli di MokaByte!
CREATE TABLE articoli
(

...,
```

```
CODICE CODART_T, — il codice articolo è di tipo CODART_T alias CHAR(8)
...
);
```

Se non esistessero i distinct type si dovrebbe aggiungere un costraint a ogni tabella che contiene il codice dell'articolo oppure definire un dominio:

```
CREATE DOMAIN CODART CHAR(8);
```

Dal punto di vista applicativo il tipo distinct type non ha corrispondenza e, per recuperare il valore di una colonna contenente un distinct type, è sufficiente far riferimento al tipo di dato originario su cui esso è definito. Nel nostro esempio quindi, per avere un banale report codice–titolo, si scriverà

```
...
ResultSet rs = stmt.executeQuery("SELECT CODICE, TITOLO FROM ARTICOLI");
System.out.println("CODICE | TITOLO");
while(rs.next()) {
  System.out.println(rs.getString("CODICE") + "\t" + rs.getString("TITOLO"));
}
...
```

Ecco ora come si utilizzano gli oggetti introdotti per manipolare i nuovi tipi SQL3.

Clob, Blob, Array

Oltre ad aver introdotto le interfacce Clob, Blob e Array, JDBC 2 ha ovviamente esteso la classe ResultSet aggiungendovi i metodi corrispondenti getClob, setClob etc., tramite i quali effettivamente manipolare i dati.

Da notare che Clob e Blob sono appunto delle interfacce, quindi non è possibile istanziare direttamente un oggetto di uno di questi tipi. Sarebbe davvero comodo se potessimo istanziare un Blob passandogli ad esempio il nome di un file contenente un'immagine da immagazzinare nel database

```
Blob b = Blob("/usr/home/nico/immagini/mokabyte.jpg");
```

e passare poi l'oggetto a un metodo setBlob() che lo inserirà nella colonna.

Esistono metodi proprietari per inserire clob e blob in una colonna. Con Java si utilizzano i metodi ResultSet.setBinaryStream() e ResultSet.setObject() per inserire blob, e il metodo ResultSet.setAsciiStream() per inserire clob.

Ecco le definizione delle interfacce Blob e Clob.

```
package java.sql;
public interface Blob {
```

```
    long length() throws SQLException;
    InputStream getBinaryStream() throws SQLException;
    byte[] getBytes(long pos, int length) throws SQLException;
    long position(byte [] pattern, long start) throws SQLException;
    long position(Blob pattern, long start) throws SQLException;
}

package java.sql;
public interface java.sql.Clob {
    long length() throws SQLException;
    InputStream getAsciiStream() throws SQLException;
    Reader getCharacterStream() throws SQLException;
    String getSubString(long pos, int length) throws SQLException;
    long position(String searchstr, long start) throws SQLException;
    long position(Clob searchstr, long start) throws SQLException;
}
```

Si supponga che nel database di MokaByte la tabella articoli abbia una colonna documento contenente l'articolo in formato MS Word. Per accedervi si utilizzano le istruzioni che seguono:

```
...
// query
String sql = "SELECT idart, titolo, doc FROM articoli, " + " WHERE idaut = 1000 AND data = '12/02/1999'";
// eseguo query
ResultSet rs = stmt.executeQuery(sql);
// so di recuperare un solo articolo!
rs.first();
String titolo = rs.getString("titolo");
// ottengo blob
Blob data = rs.getBlob("documento");
...
```

Ora, per recuperare il contenuto del documento .doc del nostro articolista e scriverlo su un file si utilizzano i metodi di Blob:

```
OutputStream doc = new BufferedOutputStream(new FileOutputStream(titolo + ".doc"));
InputStream in = blob.getBinaryStream();
byte b;
// copia dati
while ((b = in.read()) > -1) {
    doc.write();
}
// chiusura streams
```

```
doc.close();
in.close();
// log attività
System.err.println("Documento #" + rs.getString("idarticolo") + " in file " + titolo + ".doc");
```

Con i clob si opera in modo del tutto simile.

L'interfaccia Array viene utilizzata per manipolare il tipo di dati SQL ARRAY (non supportato da tutti i database). Per creare un tipo di dato ARRAY nel database si utilizza una sintassi SQL simile alla seguente:

```
CREATE TYPE INTERESSI_T AS ARRAY(10) OF VARCHAR(40);
```

Lo si utilizza poi quando si crea una tabella; ad esempio:

```
CREATE TABLE ARTICOLISTI
(
  idarticolista  SERIAL,
  nome           VARCHAR(30);
  ...
  interessi      INTERESSI_T, — array interessi (es.: 'musica', 'palestra', ...)
  ...
);
```

Si può aggiungere la colonna alla tabella, se già esiste:

```
ALTER TABLE ADD interessi INTERESSI_T;
```

È stata dunque aggiunta alla tabella articolisti del nostro database degli articoli di MokaByte una colonna interessi, di tipo ARRAY, che comprende un elenco di interessi che danno un po' di informazioni sul profilo utente dell'articolista, in particolare le sue preferenze per il tempo libero. Dunque per recuperare queste informazione dovranno essere eseguite le seguenti istruzioni:

```
...
String sql = "SELECT interessi FROM articolisti " + " WHERE nome='Nicola' AND cognome='Venditti'";
ResultSet. rs = stmt.executeQuery(sql);
Array arr = (String[])rs.getArray(1);
String interessi = arr.getArray();

System.out.println("Preferenze dell'articolista " + nome + " " + cognome + ": ");
for(int = 0; i < interessi.length(); ++i) {
  System.out.println(" " + i + ")" + interessi[i]);
}
...
```

Per l'inserimento invece occorre utilizzare la sintassi SQL per gli ARRAY.

Tipo REF

Come detto il tipo REF è un tipo di dato utilizzato per referenziare il dato vero e proprio. Il suo utilizzo è necessario per evitare inutili duplicazioni di dati. Ecco un esempio di utilizzo del tipo REF all'interno del database di MokaByte.

MokaByte organizza periodicamente dei corsi itineranti per la diffusione e l'insegnamento del linguaggio Java. Per questi corsi vengono di volta in volta "arruolati" alcuni articolisti come docenti. Si vuole tenere traccia nel database di MokaByte di tutta questa attività e per farlo introduciamo un paio di tabelle: JAVA_TOUR e CORSO_TOUR: la seconda tabella serve per raccogliere le informazioni sui corsi proposti; la prima invece, una tabella molti–a–molti associa un corso a un'articolista (che diventa così docente per quel corso) in una certa data e per un certo tour.

```
— Tabella dei corsi su Java
CREATE TABLE corsi OF CORSO (OID REF(CORSO) VALUES ARE SYSTEM GENERATED);

— Tabella dei tours
CREATE TABLE java_tours
(
   idtour INTEGER,             — un id diverso per ogni tour (non PK!)
   descr VARCHAR(200),         — descrizione di questo tour
   iddocente INTEGER,          — id articolista che si occupa di un corso nel tour
   corso REF(corsomoka),       — corso sostenuto in questo tour
   data DATE,                  — data del corso
   sponsors SPONSORS_T          — array degli sponsors
FOREIGN KEY(iddocente) REFERENCES articolisti(idarticolista)
   ON DELETE SET NULL ON UPDATE CASCADE
);
```

La tabella corsi è creata a partire dal tipo corso

```
CREATE TYPE corsomoka
(
   idcorso      INTEGER,          — id
   titolo       VARCHAR(20),      — titolo del corso
   descr        VARCHAR(200)      — descrizione corso
   costo        DECIMAL(10, 3)    — costo del corso
   argomenti    ARGOMENTI_T       — un array per gli argomenti!
);
```

A questo punto vediamo come utilizzare l'interfaccia Ref per recuperare tutti i corsi relativi al primo tour di MokaByte.

```
Statement pstmt = conn.prepareStatement("SELECT * FROM corsi WHERE oid=?");
Statement stmt = conn.createStatement();
String sql = "SELECT corso FROM java_tours WHERE nome = 'Primo Java Tour di MokaByte'";

ResultSet rs1 = stmt.executeQuery(sql);
ResultSet rs2 = null;
Ref curr =null;
while(rs1.next()) {
  curr = rs1.getRef(1);
  pstmt.setRef(1, curr);
  rs2 = pstmt.execute();
  System.out.println("");
  System.out.println("---------------------------------");
  System.out.print(rs2.getString(2));  // titolo corso
  System.out.print("---------------------------------");
  System.out.println("");
  System.out.println("Descr:" + rs2.getString(3));
  System.out.println("Costo:" + rs2.getString(4));
  System.out.print("Argomenti: "
  Array arr = rs2.getArray(5);
  String[] argomenti = (String[])arr.getArray();
  for(int i=0; i<argomenti.length;+i){
    System.out.print (argomenti[i]);
    if(i <argomenti.length) System.out.print(",");
  }
  System.out.println("");
  rs2.close();
}
```

L'esempio è puramente didattico, perché è sufficiente mettere in join le due tabelle per arrivare allo stesso risultato

```
"SELECT c.corsi FROM corsi c, java_tours j
WHERE j.nome = 'Primo Java Tour di MokaByte' AND j.oid = c.corso"
```

Struct

L'interfaccia Struct serve per mappare i tipi di dato SQL3 ROW TYPE. L'esempio più classico è quello di associare all'entità indirizzo di un database anagrafico un row type del tipo:

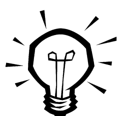
```
CREATE ROW TYPE INDIRIZZO_T  — (si utilizza anche CREATE TYPE...)
(
  citta          VARCHAR(40),
```

```
  via          VARCHAR(40),
  numero       VARCHAR(8),      — numero civico alfanumerico; p.e.: 25/A
  telefono     CHAR(10)
);
```

Ed eccone un esempio di utilizzo all'interno di una tabella.

```
CREATE TABLE articolisti
(
  idarticolista    SERIAL,        — Informix SERIAL in integer autoincrementantesi
  nome             VARCHAR(20),
…
  indirizzo        INDIRIZZO_T      — riga all'interno di una colonna!
);
```

Il row type rappresenta appunto una riga che può essere annidata, se si utilizza come nell'esempio precedente, all'interno di un'altra riga che rappresenta un'informazione strutturata che logicamente non andrebbe separata dall'entità alla quale si riferisce.



I database che includono questa caratteristica vengono detti non più semplicemente *relational database* ma *object-relational database*, anche se la qualifica *object* include poi molte altre cose.

Per trattare dati così strutturati, Java mette a disposizione l'interfaccia Struct. Ecco un esempio di utilizzo.

```
...
String sql = "SELECT indirizzo FROM articolisti " + " WHERE nome='Nicola' AND cognome='Venditti'";
ResultSet rs = stmt.executeQuery(sql);
if(rs.netx()) {
  Struct indirizzo = rs.getStruct(1);
  String[] attr = (String[])indirizzo.getAttributes();
  System.out.println("Indirizzo di Nicola Venditti: ");
  System.out.println("\t Citta: " + attr[1]);
  System.out.println("\t Via: " + attr[2]);
  System.out.println("\t Telefono: " attr[4]);
}
...
```

Il metodo Struct.getAttributes() restituisce tutti i campi in formato di array di oggetti.

Serializzazione degli oggetti Java nel database

L'introduzione del tipo Struct permette di utilizzare i tipi di dati complessi: siamo quasi a un passo da un oggetto vero e proprio. Perché ci sia una vera corrispondenza logica tra tipo strutturato SQL (row type) e tipo Java occorrerebbe che l'oggetto restituito da ResultSet.getXXX() fosse direttamente un oggetto Java di classe adeguata; ad esempio, per continuare con l'esempio sopra riportato, occorrerebbe che il result set restituisse un oggetto di classe Indirizzo con attributi simili a quelli di INDIRIZZO_T.

Questa possibilità esiste ed è indicata dalla documentazione del JDBC come *custom mapping*. È possibile cioè indicare al driver JDBC quale classe si vuole utilizzare e si vuole che sia restituita in corrispondenza di un dato strutturato. Fatto ciò è sufficiente una chiamata al metodo ResultSet.getObject() per ottenere un riferimento all'oggetto della classe di mapping e una conversione di cast per avere l'oggetto finale. Traduciamo quanto appena detto in un esempio:

```
...
Connection conn = DriverManager.getConnection(user, passwd);

// ottiene mappa tipi
java.util.Map map = conn.getTypeMap();

// aggiunge il tipo personalizzato INDIRIZZO_T
map.put("'mokabyte'.INDIRIZZO_T", Class.forName("Address");
...
String sql = "SELECT indirizzo FROM articolisti " + " WHERE nome='Nicola' AND cognome='Venditti'";
ResultSet rs = stmt.executeQuery(sql);
  rs.next();

// recupera il row type come oggetto Indirizzo e non come
// generica Struct
Indirizzo indirizzo = (Indirizzo)rs.getObject("indirizzo");
...
```

Il codice apre la connessione, ottiene un riferimento alla mappa di corrispondenza dei tipi (tipo SQL <—> tipo Java), aggiunge un'entry per il tipo INDIRIZZO_T e infine utilizza getObject() e un semplice cast per ricavare l'oggetto indirizzo di tipo Indirizzo.

La classe Java Indirizzo non può essere scritta liberamente ma deve sottostare ad alcune regole. In particolare, deve implementare l'interfaccia java.sql.SQLData di JDBC 2. Per la verità ogni DBMS ha uno strumento che, a partire dal tipo row type nel database, genera il codice per una classe di mapping: questo può semplificare non poco il lavoro.

Nel nostro caso la classe potrebbe essere scritta così:

```
public class Indirizzo implements SQLData {
  public String citta;
  public String via;
  public int numero;
```

```
    public int telefono;
    private String sql_type;
    public String getSQLTypeName() {
      return sql_type;
    }

    public void readSQL(SQLInput stream, String type) throws SQLException {
      sql_type = type;
      citta = stream.readString();
      via = stream.readString();
      numero = stream.readInt();
      telefono = stream.readString();
    }

    public void writeSQL(SQLOutput stream) throws SQLException {
      stream.writeString(citta);
      stream.writeString(via);
      stream.writeInt(numero);
      stream.writeString(telefono);
    }
  }
```

La mappa dei tipi così definita è a livello di connessione, quindi tutti i metodi utilizzeranno questa per le conversioni di tipo; tuttavia esiste anche la possibilità di definire mappe alternative da passare ad alcuni metodi che supportano questa caratteristica.

Estensioni di JDBC

Durante lo sviluppo di JDBC 2.0 ci si rese conto che, per varie ragioni, era opportuno dividere l'intera API in due parti: una parte fu chiamata JDBC Core API e l'altra JDBC Extension API (poi Optional Package). L'intenzione dei progettisti di JDBC era di separare quella parte dell'API che formava il nucleo o *core*, da quelle parti che venivano aggiunte per soddisfare le esigenze di integrazione con la sempre crescente J2EE da un lato e la richiesta di nuove funzionalità da parte della comunità degli sviluppatori dall'altro. Per evitare di dover rilasciare una nuova versione di JDBC ogni volta che si aggiungono nuove funzionalità o nuovi moduli, si è scelto di seguire questa strada: di separare ciò che è ormai consolidato (parliamo per lo più di un'API per tutti i database di tipo OLTP) da ciò che costituisce solo un'aggiunta o un arricchimento del core. Cosa succede nel caso ad esempio si voglia aggiungere il supporto per il data-warehousing in JDBC? Semplice: verrà creato un package aggiuntivo (una Standard Extension quindi) diciamo javax.olap.*, che conterrà tutta le classi utili per accedere e trattare con i data-warehouse. Attualmente, dopo diverse naming convention, esistono due parti di JDBC, perfettamente integrate in J2SE:

- il package java.sql.*, che contiene il Core API JDBC;
- il package javax.sql.* che contiene le estensioni (ex Standard Extension, ex Optional Package).

Il secondo è oggetto di questo capitolo che conserva la dicitura "Estensione di JDBC o Estensioni Standard di JDBC" solo per comodità.

Estensioni di JDBC (javax.sql.*)

Per estensioni di JDBC base s'intendono le seguenti tecnologie:

- JDBC Data Sources
- Connection Pooling
- Transazioni distribuite
- Rowset

Tranne i Rowset le altre estensioni sono state principalmente introdotte per ottenere una maggiore integrazione con J2EE (Java 2 Enterprise Edition).

La prima, JDBC Data Source, consiste in una integrazione di JDBC con il servizio dei nomi di JNDI di J2EE, per fare in modo che il database sia raggiungibile con un nome e le applicazioni Java si riferiscano ad esso con questo nome logico, guadagnando così un grado di astrazione che permette alle applicazioni stesse di essere del tutto portabili tra due piattaforme J2EE compliant.

Il meccanismo del Connection Pooling, che è stato sicuramente molto apprezzato da tutti gli sviluppatori, dà la possibilità di utilizzare un componente intermedio per l'accesso al database che si occupa di gestire le connessioni (in tal senso è un Connection Manager), pre-allocandone un certo numero e distribuendole (pooling) su richiesta ai client.

L'estensione di JDBC per il supporto delle transazioni distribuite nasce anche dall'esigenza di integrazione con un altro componente di J2EE e precisamente l'interfaccia Java Transaction API, che permette alle applicazioni di avviare una transazione su più database che supportano questa funzionalità. Le transazioni distribuite estendono il concetto di transazione SQL, permettendo di fare operazioni (con relativo rollback) su più database, anche di diversi DB vendor. Occorre solo che supportino in modo nativo il protocollo XA per le transazioni distribuite, oltre a fornirne un'implementazione attraverso il loro driver JDBC.

L'ultima estensione trattata è quella dei rowset che dà la possibilità di avere degli oggetti che somigliano a dei result set, ma possono ad esempio essere disconnessi dal database, e che in più permettono una maggiore integrazione dei dati nel modello a componenti basato su JavaBeans.

Ecco le interfacce introdotte per supportare le funzionalità appena discusse, da notare come siano tutte poste in package del tipo javax.sql* cioè appunto package di estensione della VM.

```
javax.sql.DataSource

javax.sql.XAConnection
javax.sql.XADataSource

javax.sql.ConnectionEvent
javax.sql.ConnectionEventListener
javax.sql.ConnectionPoolDataSource
javax.sql.PooledConnection

javax.sql.RowSet
javax.sql.RowSetEvent
javax.sql.RowSetInternal
javax.sql.RowSetListener
javax.sql.RowSetMetaData
javax.sql.RowSetReader
javax.sql.RowSetWriter
```

Si vedrà ora un po' più in dettaglio quanto introdotto, a partire dal supporto per JNDI.

JNDI Data Source

Visto che l'argomento è già stato introdotto, in questa sezione si vedrà come si utilizza JNDI per accedere al database.

JNDI è una delle enterprise API fornite da J2EE che permette alle applicazioni di accedere a servizi standard dalla piattaforma. JNDI permette di accedere al servizio di nomi e di directory che permette a sua volta di raggiungere risorse di vario tipo attraverso i due diffusi meccanismi di lookup e browsing, due metodi di ricerca, il primo dei quali utilizzato quando si conosce il nome della risorsa (o almeno un pattern per il nome), il secondo invece utilizzato quando si conoscono degli attributi della risorsa.

Si torni per un attimo alla maniera tradizionale in cui una applicazione accede a una sorgente di dati. Per prima cosa si carica il driver JDBC:

```
Class.forName("com.informix.jdbc.IfxDriver");
```

poi viene creata la connessione (nell'esempio a un database Informix),

```
Connection con = DriverManager.getConnection("jdbc:informix-sqli://dbserver.mokabyte.it:1526/mokadb:INFORMIXSERVER
                                              =ol_mokabyte;USER=mokauser;PASSWORD=mokaifx");
```

il resto è infine normale programmazione JDBC.

Questo metodo comporta la conseguenza evidente che occorre sapere la locazione del database a cui ci si connette, ma se cambia per qualche ragione anche solo la porta su cui

ascolta il database server bisogna rimettere mano al codice. Inoltre si fa esplicito riferimento alla classe del driver JDBC da utilizzare.

Certo bisogna modificare quasi sempre solo una linea di codice, o un file di configurazione, ma qui si vuole evidenziare che manca quel passaggio intermedio per togliere ogni riferimento alla base di dati vera e propria e per disaccoppiare ciò che è programmazione e ciò che è amministrazione (gestione dei driver, dei socket, etc.).

Perché si possa raggiungere questa separazione occorre che le applicazioni possano contare su un ambiente esterno che offra dei servizi come il JNDI, per mezzo del quale un nome può rappresentare una sorgente di dati. L'applicazione continuerà a utilizzare lo stesso nome (per esempio `MokaDB`) per connettersi alla base dati sulla quale opera. Si fa in modo insomma di demandare alla fase di deploy quegli aspetti preliminari e di contorno non legati alla logica applicativa.

Ecco come ci si connette a una base di dati utilizzando il nome ad essa associato, in un'applicazione J2EE.

```
...
javax.naming.Context ctx = new javax.naming.InitialContext();
javax.sql.DataSource ds =(javax.sql.DataSource)ctx.lookup("jdbc/mokadb");
java.sql.Connection conn = ds.getConnection("mokar", "corretto");
...
```

Nessun driver da precaricare, nessun riferimento a parametri fisici come il dbserver o la porta del socket.

Per prima cosa si crea un `Context` per accedere al servizio dei nomi, poi si ottiene un'interfaccia `DataSource` verso la base dati e, infine, si ottiene la connessione attraverso il metodo `DataSource.getConnection()`. Per funzionare, tutto questo ha bisogno di un lavoro preliminare che prepari l'ambiente in cui l'applicazione sarà installata ed eseguita.

Questa fase preliminare si riassume in tre passi:

- creazione di un oggetto `DataSource`;
- impostazione delle proprietà del `DataSource`;
- registrazione del `DataSource` presso un servizio dei nomi (per lo più JNDI in J2EE).

Ecco del codice che esegue il primo e il secondo compito, supponendo di avere un nostro driver JDBC che supporta il meccanismo dei `DataSource`:

```
...
it.mokabyte.MokaDataSource ds = new it.mokabyte.MokaDataSource();
ds.setServerName("ol_mokabyte");
ds.setDatabaseName("mokadb");
ds.setDescription("Database di tutti gli articoli di MokaByte");
...
```

Adesso la parte più importante, la registrazione della nostra fonte di dati presso il servizio dei nomi utilizzando il nome mokadb, che tutte le applicazioni client utilizzeranno per riferirsi al database degli articoli:

```
…
Context ctx = new InitialContext();
ctx.bind("jdbc/mokadb", ds);
…
```

Il nome completo della risorsa è "jdbc/mokadb" è non solo mokadb, perché JNDI ha una struttura gerarchica dei nomi, e quindi occorre in generale l'intero percorso per riferirsi a una risorsa, altrimenti il naming service non riuscirà a trovare ciò che stiamo cercando.

Per concludere questa sezione ancora due parole sugli oggetti DataSource. Vi sono diverse possibilità o per meglio dire livelli di implementazione di un oggetto DataSource:

- una implementazione di base di DataSource che fornisce oggetti standard Connection non soggetti a pooling né utilizzabili in transazioni distribuite;

- una implementazione della classe DataSource che supporta il connection pooling, producendo oggetti Connection riciclabili nel senso del Connection Pool Manager: l'interfaccia interessata è ConnectionPoolDataSource;

- una implementazione completa di DataSource che supporta rispetto alla precedente anche le transazioni distribuite, fornendo oggetti Connection utilizzabili in una transazione distribuita: l'interfaccia interessata è XADataSource.

Nel caso di pooled DataSource si userà codice simile al seguente:

- per registrare il Connection Pool Manager…

```
// Crea il Pool Manager
it.mokabyte.ConnectionPoolDataSource cpds = new it.mokabyte.ConnectionPoolDataSource();

// Imposta le proprietà
cpds.setServerName("ol_mokabyte");
cpds.setDatabaseName("mokadb");
cpds.setPortNumber(1526);
cpds.setDescription("Connection pooling per il database degli articoli di MokaByte.");
// Registra presso il servizio dei nomi il Pool Manager
Context ctx = new InitialContext();
ctx.bind("jdbc/pool/mokadbCP", cpds);
```

- per registrare il DataSource…

```
it.mokabyte.PooledDataSource ds = new it.mokabyte.PooledDataSource();
ds.setDescription("Pooled Connections per il DB degli articoli di MokaByte");
ds.setDataSourceName("jdbc/pool/mokadbCP");
Context ctx = new InitialContext();
ctx.bind("jdbc/mokadb", ds);
```

Nel caso invece di un DataSource che supporta pooling e transazioni distribuite si userà il seguente codice:

- per registrare il Distributed Transaction DataSource…

```
// istanzia oggetto che implementa XADataSource
it.mokabyte.XATransactionalDS xads = new it.mokabyte.XATransactionalDS();
// impostazioni
xads.setServerName("mokadb");
xads.setDatabaseName("mokadb");
xads.setPortNumber(1526);
xads.setDescription("Manager per transazioni distribuite");
// registrazione
Context ctx = new InitialContext();
ctx.bind("jdbc/xa/mokadbXA", xads);
```

- per registrare il DataSource…

```
// istanzia oggetto che implementa DataSource
it.mokabyte.TransactionalDataSource ds = new it.mokabyte.TransactionalDataSource();
// impostazioni
ds.setDescription("DataSource per Transazioni Distribuite");
ds.setDataSourceName("jdbc/xa/mokadbXA");
// binding della risorsa al nome logico
Context ctx = new InitialContext();
ctx.bind("jdbc/mokadb", ds);
```

Gli oggetti di classe it.mokabyte.PooledDataSource e it.mokabyte.TransactionalDataSource che implementano l'interfaccia DataSource sono ovviamente studiati per lavorare con i corrispettivi oggetti di tipo it.mokabyte.ConnectionPoolDataSource e it.mokabyte.XATransactionalDS, che implementano a loro volta le interfacce più specializzate ConnectionPoolDataSource e XADataSource.

Concludendo, si può dire che, dove possibile, un'applicazione Java dovrebbe utilizzare un DataSource invece del tradizionale metodo basato su DriverManager per accedere ai dati, per guadagnare in portabilità ed eleganza definendo gli aspetti più prettamente sistemistici durante la fase di deploy.

Connection Pooling

Nella precedente sezione si è visto come registrare un pooled DataSource e come utilizzarlo. Si è visto, quindi, che dal punto di vista dell'applicazione il pooling delle connessioni è del tutto trasparente.

Un Connection Pool è una cache di connessioni a database gestita da un componente, detto Connection Manager, solitamente fornito dal JDBC Driver Vendor insieme al driver stesso, che si occupa di recuperare da tale cache una connessione quando un client ne faccia richiesta e deallocarla quando un client non ne ha più bisogno.

Dato che il meccanismo di pooling è del tutto trasparente per le applicazioni, come fa il connection manager a sapere quando entrare in azione? Si dia uno sguardo alla seguente porzione di codice che utilizza un DataSource per accedere ai dati:

```
...
// Si ottiene una connessione da un pooled DataSource
// preliminarmente ottenuto in qualche maniera
Connection con = ds.getConnection("jdbc/mokadb", "mokauser", "mokaifx");

// Si eseguono un po' di operazioni SQL sul DB degli
// articoli, eventualmente disabilitando l'autocommit
...

// Chiude la connessione
con.close();
...
```

Quando si invoca il metodo DataSource.getConnection() non viene aperta una connessione come quando si invoca DriverManager.getConnection() ma, nel caso di pooled DataSource, viene restituito un riferimento a una connessione esistente e attiva nel pool di connessioni predefinite, evitando al database e alla JVM l'overhead di allocazione delle risorse e il setup di una nuova sessione. La connessione rimarrà allocata fin quando il client non termina il lavoro sul database.

Quando quindi, nell'ultima istruzione, il client decide di disconnettersi dal database o più propriamente decide di terminare la sessione di lavoro, la chiamata Connection.close() viene gestita dal pool manager che dealloca la connessione e la rende disponibile per altre applicazioni nella cache.

Transazioni distribuite

Il protocollo di transazione distribuita tra database XA esiste già da tempo. Dato che è un elemento ricorrente dell'enterprise computing, Sun ha pensato bene di includerlo nella sua piattaforma J2EE attraverso le due API JTA (Java Transaction API) e JTS (Java Transaction Service). Tale aggiunta non poteva non avere impatto ovviamente su JDBC: è stata così introdotta questa estensione standard per JDBC che permette di vedere il database come una risor-

sa abilitata alla transazione distribuita (posto che ovviamente il sottostante driver di database abbia già questa caratteristica).

Anche nel caso di transazioni distribuite il programmatore JDBC ha poco da scrivere, visto che per lui è trasparente il fatto che le operazioni SQL che la sua applicazione esegue facciano parte di una transazione distribuita. Più che altro in questo caso occorre sapere che vi sono delle restrizioni rispetto al caso di accesso a un solo database. Precisamente bisogna evitare di chiamare i metodi Connection.setAutoCommit(), Connection.commit() e Connection.rollback(). E ciò perché questi metodi hanno un controllo diretto sulla transazione in corso che però viene gestita dal Transaction Manager del middle-tier server (solitamente è un application server).

Rowset

Il rowset è un oggetto definito per rappresentare un set di righe di database. Un oggetto di questo tipo implementa l'interfaccia javax.sql.RowSet che a sua volta estende java.sql.ResultSet. L'interfaccia RowSet è stata disegnata con in mente il modello a componenti di JavaBeans.

In tal senso i rowset hanno proprietà, a cui si accede con il tradizionale pattern get/set. Ad esempio:

```
...
// imposta la sorgente dati
rowset.setDataSourceName("jdbc/mokadb");
// definisce il livello di isolamento
rowset.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
// definisce query da eseguire
rowset.setCommand("SELECT NOME FROM ARTICOLISTI WHERE id=2");
// esegue il comando SQL
rowset.execute();
...
```

I rowset supportano come ogni bean anche gli eventi: il meccanismo di gestione è lo stesso: permette a listener di registrarsi attraverso RowSet.addRowSetListener() e di ricevere, nel caso si verifichi un evento, una notifica e un oggetto RowSetEvent con informazioni sull'evento appena occorso.

Sun ha definito per i rowset solo delle interfacce che ne definiscono il modello e la logica di funzionamento (si tratta cioè una specifica). L'implementazione è lasciata al produttore del database, che definirà la logica implementativa attraverso oggetti che implementano queste interfacce.

Sun ha anche individuato tre possibili e naturali implementazioni di rowset: il CachedRowSet, il JDBCRowSet, il WebRowSet.

CachedRowSet

Il CashedRowSet è un contenitore tabulare di dati disconnesso, serializzabile e scrollabile. Queste proprietà lo rendono adatto a dispositivi disconnessi o solo occasionalmente connessi come i portatili, i palmari e altri dispositivi simili.

Un rowset così strutturato permette a tali dispositivi di poter lavorare sui dati in locale senza connessione al server, come se si avesse una implementazione locale di JDBC e una locale sorgente di dati, anche se in realtà i dati sono stati recuperati e immagazzinati nel rowset in precedenza.

Per popolare un CachedRowSet prima di spedirlo, ad esempio, a un thin client che vi lavorerà, si utilizza del codice come il seguente:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM ARTICOLI");
CachedRowSet crset = new CachedRowSet();
crset.populate(rs);
```

Dopo aver popolato il rowset si può chiudere la connessione a cui si riferisce lo statement stmt (non mostrato) e spedire l'oggetto crset attraverso la rete a un client che può elaborarlo a piacimento prima di apportare le modifiche definitive al database. È ovvio in questo il vantaggio per il client della possibilità di lavorare sui dati come se si avesse a disposizione un result set di JDBC 2, ma senza la necessità di una connessione aperta al database.

Il metodo alternativo per popolare il CachedRowSet è rappresentato da CachedRowSet.execute(), che esegue direttamente la query al database senza passare per un oggetto ResultSet.

Il CachedRowSet, dal punto di vista del client è in tutto e per tutto assimilabile a un ResultSet: l'applicazione client utilizzerà i metodi già visti per accedere e modificare i dati e solo quando si avrà la certezza di poter sottoporre le modifiche al database attraverso il metodo CachedRowSet.acceptChanges(), il CachedRowSet aprirà la connessione al database utilizzando come parametri le proprietà del CachedRowSet stesso e, se i dati originali nel frattempo non sono cambiati (ma è una scelta dell'implementatore), le modifiche avranno luogo.

È possibile recuperare lo stato originale dei dati con il metodo CachedRowSet.restoreOriginal().

JDBCRowSet

Questo tipo di rowset è poco più di un layer intorno all'oggetto ResultSet con cui l'applicazione JDBC può accedere ai dati. Il vantaggio di questo wrapping è che il JDBCRowSet è di fatto un componente JavaBeans è quindi può essere inquadrato in un modello a componenti per la propria applicazione.

WebRowSet

Anche il WebRowSet è una specializzazione del RowSet che risolve un problema specifico.

Esso viene utilizzato per trasferire i dati di un ResultSet a client fuori dal firewall aziendale, e che quindi possono al più utilizzare il protocollo HTTP per comunicare con il DB server. Questo rowset utilizza al proprio interno il protocollo HTTP per il tunneling della connessione al database e per comunicare con una servlet sul web server, possibilmente utilizzando XML.

Schema del database di demo utilizzato dal codice di esempio (in crea_mokadb.sql)

```
--
-- Crea l'utente e gli assegna i permessi necessari
--
CREATE USER moka IDENTIFIED BY corretto;
GRANT resource, connect TO moka;

--
-- Si riconnette con l'utente moka per
-- creare gli oggetti nel proprio schema.
--
CONNECT moka/corretto;


prompt ^^ Creazione tabella lettori
CREATE TABLE lettori(
  iduser         INTEGER          CONSTRAINT pk_iduser PRIMARY KEY NOT NULL,
  nome           VARCHAR2(30)     NOT NULL,
  cognome        VARCHAR2(40)     NOT NULL,
  email          VARCHAR2(30),
  email_uff      VARCHAR2(30),
  indirizzo      VARCHAR2(200),
  indirizzo_uff  VARCHAR2(200),
  data           DATE,
  non_pubblico   CHAR,
  CONSTRAINT uq_email_lett        UNIQUE(email),
  CONSTRAINT ck_non_pubblico      CHECK(non_pubblico in ('S','N'))
  );

CREATE SEQUENCE sq_lettori;


prompt ^^ Creazione tabella servizi
CREATE TABLE servizi(
  idserv       INTEGER          CONSTRAINT pk_idservPRIMARY KEY NOT NULL,
  nome         VARCHAR2(30)     NOT NULL,
  data_avvio   DATE             NOT NULL,
  descr        VARCHAR2(40),
  prezzo       DECIMAL(2,5),
```

```
  stato           VARCHAR2(20),
  note            VARCHAR2(50),
  CONSTRAINT uq_nome      UNIQUE(nome),
  CONSTRAINT ck_stato     CHECK(stato in ('ATTIVO', 'TERMINATO', 'SOSPESO'))
  );

CREATE SEQUENCE sq_servizi;

prompt ^^ Creazione tabella sottoscrizioni
CREATE TABLE sottoscrizioni(
  iduser   INTEGER     REFERENCES lettori(iduser),
  idserv   INTEGER     REFERENCES servizi(idserv),
  data     DATE        NOT NULL,
  CONSTRAINT pk_sottosc    PRIMARY KEY (iduser, idserv)
  );


prompt ^^ Creazione viste

-- Vista delle sole sottoscrizioni degli utenti pubblici
CREATE VIEW lettori_moka
  AS SELECT   *
    FROM      lettori
    WHERE     non_pubblico != 1;

-- GRANT SELECT ON lettori_moka TO  PUBLIC;

-- Vista delle sole sottoscrizioni degli utenti pubblici
CREATE VIEW sottoscrizioni_moka
  AS SELECT   l.iduser,
        l.nome,
        l.cognome,
        v.nome     AS NomeServizio,
        v.descr    AS DscrizioneServizio
    FROM   sottoscrizioni s,
        lettori l,
        servizi v
    WHERE  s.iduser = l.iduser
        AND s.idserv = v.idserv
        AND l.iduser != 1;

--
-- Tabella degli autori di Mokabyte
```

```
--

CREATE TABLE autori(
   idaut        INTEGER          CONSTRAINT pk_idaut PRIMARY KEY NOT NULL,
   nome        VARCHAR2(30)   NOT NULL,
   cognome      VARCHAR2(40)     NOT NULL,
   email          VARCHAR2(30),
   societa        VARCHAR2(100),
   indirizzo       VARCHAR2(200),
   data            DATE,
   area            VARCHAR(200),
   note            VARCHAR(200),
   mokabook    CHAR,
   CONSTRAINT uq_email_auth   UNIQUE(email),
   CONSTRAINT ck_mokabook    CHECK(mokabook in ('S','N'))
   );

CREATE SEQUENCE sq_autori;

--
-- Tabella degli articoli di mokabyte
--
CREATE TABLE articoli(
   idart        INTEGER      CONSTRAINT pk_idart PRIMARY KEY NOT NULL,
   idaut        INTEGER      CONSTRAINT fk_idaut REFERENCES autori(idaut),
   titolo        VARCHAR2(200) NOT NULL,
   abstract    CLOB,
   testo        CLOB,
   link          CLOB,
   doc          BLOB,
   data         DATE
   );

CREATE SEQUENCE sq_articoli;

QUIT
```

Lo script SQL va eseguito come segue:

```
<prompt>  sqlplus "user/pwd as sysdba"  @crea_mokadb.sql
```

dove user e pwd sono rispettivamente username e password di un utente di sistema del gruppo SYSDBA (per esempio SYS).

Capitolo 3

Java e internazionalizzazione

GIOVANNI PULITI

Introduzione

Con l'avvento di Internet il mondo ha ridotto notevolmente le sue dimensioni. Ogni pagina web è distante un solo click da una qualsiasi altra, offrendo da un lato un numero incredibile di possibilità, ma ponendo al contempo una serie di problematiche nuove.

Si pensi ad esempio alle applet e alle applicazioni web: in questo caso la possibilità di fare eseguire una applicazione in una qualsiasi parte del mondo introduce il problema legato alla lingua utilizzata nella descrizione dell'interfaccia grafica o alle regole grammaticali in uso in quella regione geografica.

Anche se il problema non è nuovo, con l'avvento della rete si deve tener conto del fattore tempo, dato che la medesima applicazione può essere eseguita nel medesimo istante in luoghi differenti.

Il concetto di portabilità non è più quindi esclusivamente legato ai dettagli tecnologici della piattaforma utilizzata, ma deve tener conto anche di tutti quei formalismi in uso nel Paese di esecuzione per la rappresentazione di testi, valute, date e numeri.

Nasce quindi l'esigenza di poter disporre di strumenti che offrano la possibilità di scrivere applicazioni in modo indipendente dalla cultura di coloro che le andranno a utilizzare.

Il processo volto alla creazione di programmi slegati dal particolare contesto linguistico e grammaticale in uso nel momento dell'utilizzo del software, è detto localizzazione e internazionalizzazione di una applicazione.

La localizzazione permette l'adattamento del software a una particolare regione aggiungendo all'applicazione tutte quelle parti che dipendono dalla regione stessa, come ad esempio le scritte di una interfaccia grafica.

L'internazionalizzazione invece consente a una applicazione di adattarsi in modo automatico alle convenzioni in uso in una particolare lingua o regione senza che si debba ricorrere alla

riscrittura o, peggio, alla riprogettazione dell'applicazione stessa: si pensi in questo caso alle differenti modalità di rappresentazione dei numeri o delle date.



Dato che molte volte risulta piuttosto scomodo utilizzare per intero la parola *internationalization* (si pensi ad esempio ai nomi delle directory), essa viene spesso indicata con la sigla *i18n*, dato che nella parola *internationalization*, sono presenti 18 caratteri fra la prima lettera *i* e l'ultima lettera *n*. La localizzazione, sempre per lo stesso motivo, viene abbreviata con la sigla *l10n*.

Per brevità quando ci si riferisce alla internazionalizzazione si intendono entrambi gli aspetti, ed è anzi piuttosto raro sentire parlare di *localization*. Si tenga presente che nella maggiore parte dei casi la *internationalization* non rappresenta l'unica soluzione ai problemi derivanti dalla localizzazione, ma offre notevoli vantaggi in termini di praticità e semplificazione. Anche in questo caso non sono state aggiunte particolari innovazioni tecnologiche rispetto al passato, limitandosi invece a mettere ordine e contestualizzare il lavoro e l'esperienza fatta in passato con altri linguaggi e tecnologie.

L'interfaccia grafica internazionale

Nel caso in cui un determinato software debba soddisfare i requisiti di *locale-independent* la prima problematica che si deve cercare di risolvere è la traduzione automatica delle varie scritte e messaggi che compongono l'interfaccia grafica.

Si consideri ad esempio il caso della label di un pulsante di conferma in una finestra di dialogo: nei Paesi di lingua inglese potrà essere utilizzata l'etichetta `OK`, composta da due caratteri, mentre se l'applicazione deve essere venduta in Italia, si dovrebbe utilizzare la stringa `"CONFERMA"`, costituita invece da otto caratteri. In questo caso quindi, senza gli strumenti del package `java.text` si dovrebbe creare un pulsante per l'applicazione inglese ed uno per quella italiana con dimensioni diverse. Se inoltre il bottone è un componente bean allora la filosofia di base della riutilizzabilità viene meno.

Data entry

Una corretta interazione con l'utente finale deve tener conto anche della fase di inserimento dati, dato che una fonte d'errori consiste nella errata valutazione del contesto esatto di frasi e dati in genere.

Le API relative all'Internationalization non risolvono completamente il problema, ma permettono di dividere in maniera decisa l'implementazione delle classi dal linguaggio che esse supportano. I vantaggi di questa scelta emergono quando si devono estendere tali classi o aggiungere linguaggi nuovi.

Un esempio tipico è dato dalla tastiera utilizzata: le tastiere italiane infatti sono diverse da quelle giapponesi, e in questo caso i caratteri a disposizione di un giapponese (con PC abilita-

to a tale lingua) sono molti di più di quelli a disposizione di un italiano. Inoltre in Giappone per visualizzare un carattere non è sufficiente premere un tasto ma occorre il più delle volte una loro combinazione.

Java, fin dalla prima versione, prevede la gestione del formato Unicode che contiene quasi tutti i caratteri utilizzati nel mondo e quindi il problema non è la disponibilità dei caratteri ma la conoscenza di quale font utilizzare. Quando Java inizializza lo stream di input (System.in) viene letto il file font.properties nella directory \lib relativa a quella del JDK.

Questo file organizzato secondo il formato dei file Properties, contiene le informazioni sul font utilizzato per l'input. Nel caso dell'Italia l'ultima riga di tale file è la seguente

```
# charset for text input
#
inputtextcharset = ANSI_CHARSET
```

in cui si specifica che il set di caratteri utilizzato è quello ANSI. Nel caso del Giappone il file font.properties sarebbe stato quello che nel nostro Paese prende il nome font.properties.ja. In questo file l'ultima riga è la seguente

```
# charset for text input
#
inputtextcharset = SHIFTJIS_CHARSET
```

in cui si indica che il font per l'inserimento dei caratteri è quello giapponese.

Si tenga presente che per utilizzare caratteri giapponesi in un programma Java non è sufficiente modificare le ultime righe del file font.properties dato che anche il sistema operativo in uso deve essere abilitato a tale font. Il problema legato al punto precedente è la varietà dei font intesi come caratteri e non come corrispondenza carattere–tasto.

Oltre la sintassi

Oltre alla corrispondenza dei caratteri, esistono problemi di corrispondenza nella struttura di una medesima frase nell'ambito di due contesti linguistici differenti: si supponga ad esempio di dover stampare un messaggio funzione di una determinata variabile come nel caso seguente.

```
System.out.println("La torre è alta " + res + " metri");
```

Si potrebbe anche scrivere

```
System.out.println(res + " metri è l'altezza della torre");
```

Il problema è che non tutte le lingue seguono la struttura soggetto–predicato–complemento, ma possono avere strutture più complesse in cui, come nel caso del cinese o del turco, l'ordine delle parole nella frase è molto significativo.

La gestione della struttura della frase e dei messaggi utilizzati nell'applicazione è un tipo di compito più complesso e verrà visto nella parte finale del capitolo.

La gestione delle font

Java ha adottato da subito lo standard Unicode. Nel JDK 1.0 si utilizza lo Unicode 1.1 mentre nella versione 1.1 si utilizza lo Unicode 2.0 che prevede la gestione di ben 38 885 caratteri derivanti dalla unione di quelli utilizzati da 25 culture diverse: i caratteri utilizzati dalla lingua italiana sono compresi tra \u0020 e \u007E. Lo Unicode però, pur essendo un formato universale, non sempre viene utilizzato per la codifica dei caratteri; a seconda del caso particolare, la preferenza viene accordata ad altri formati.

Un formato molto utilizzato ad esempio è l'UTF-8 (Uniform Text Format a 8 bit) che permette di rappresentare ciascun carattere Unicode in un numero variabile di byte a seconda del carattere da rappresentare, essendo questo un codice a lunghezza variabile. Alcuni caratteri sono rappresentati con un unico byte, altri con due e altri con tre. Quelli rappresentati con un unico byte coincidono con quelli ASCII. Ciascun byte utilizza i primi bit come identificatori della sua posizione all'interno della codifica del carattere.

L'UTF-8 viene spesso utilizzato come formato di codifica delle stringhe durante le trasmissioni dato che, tra l'altro, è compatibile sui sistemi Unix. Utilizzando un editor ASCII, come quello utilizzato ad esempio dalla maggior parte degli ambienti di sviluppo, per inserire un carattere Unicode non ASCII è necessario ricorrere alla sua codifica.

Per convertire una stringa da formato Unicode a UTF-8 si può operare come nel programma Utf2Unicode: supponendo di voler rappresentare la stringa "perché", si può scrivere

```
String unicodeString = new String("perch " + "\u00E9");
```

Allora tramite il metodo getBytes(), specificando il formato è possibile ricavare un array di caratteri in formato UTF-8; ad esempio

```
byte[] utf8Bytes = original.getBytes("UTF8");
```

mentre

```
byte[] unicodeBytes = original.getBytes();
```

permette di ricavare un array analogo, ma contenente caratteri in formato Unicode.

Effettuando la stampa di entrambi gli array ci si potrà rendere conto effettivamente di come l'UTF-8 sia un formato a codifica variabile.

Per riottenere una stringa Unicode partendo dal formato UTF-8 è sufficiente specificare tale informazione nel costruttore della String, ovvero

```
String unicodeString2 = new String(utf8Bytes, "UTF8");
```

Si è detto che Java gestisce l'Unicode internamente trasformando ogni carattere proprio della cultura locale nel corrispondente carattere Unicode. Nella versione JDK 1.0 non era possibile agire su questa conversione mentre dalla versione JDK 1.1 è possibile farlo indirettamente attraverso le due classi InputStreamReader e OutputStreamWriter contenute nel package java.io.

Queste due classi, facendo uso di due particolari classi del package sun.io ottengono la conversione da Byte a Char e viceversa, in maniera *locale sensitive* cioè utilizzando i caratteri locali.

Tali classi devono poter disporre di un meccanismo per conoscere la regione in cui sono in esecuzione in modo da produrre la giusta conversione e infatti utilizzano la proprietà file.encoding delle Properties di sistema.

Nella tab. 3.1 sono riportati alcuni dei valori possibili (gli altri sono disponibili nella documentazione del JDK 1.1).

Nella classe ProvaFileEncoding viene brevemente mostrato come utilizzare tali classi per la conversione. Il cuore di questo semplice programma è il seguente:

```
// Esegue la conversione
PipedInputStream pis = new PipedInputStream();
PipedOutputStream pos = new PipedOutputStream(pis);
InputStreamReader isr = new InputStreamReader(pis, txf_encoding.getText());
BufferedReader br = new BufferedReader(isr);
OutputStreamWriter osr = new OutputStreamWriter(pos, "8859_1")
BufferedWriter bw = new BufferedWriter(osr);

// Scrive sul writer
int lunghezza = txf_testo.getText().length();
char[] caratteri = new char[lunghezza];
txf_testo.getText().getChars(0, lunghezza, caratteri, 0);
```

Figura 3.1 – *Processo di conversione degli stream di caratteri da non Unicode a Unicode e viceversa.*

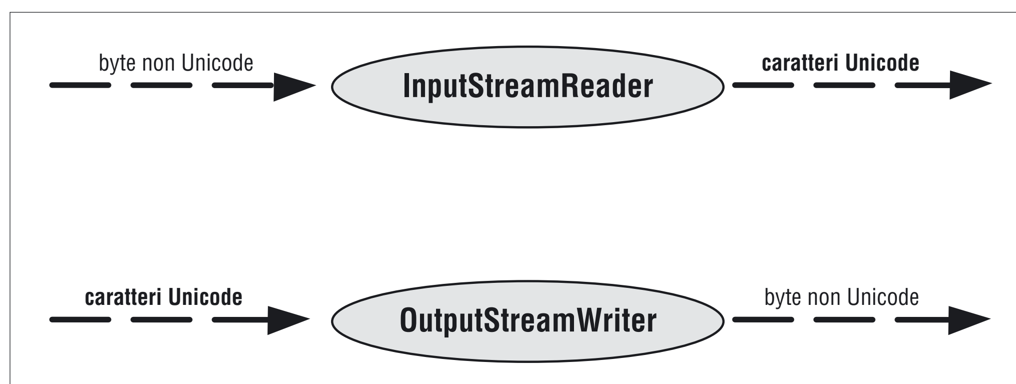


Tabella 3.1 – *Descrizione della codifica del file.encoding.*

Proprietà file.encoding	Descrizione della codifica
8859_1	ISO Latin-1
8859_2	ISO Latin-2
8859_3	ISO Latin-3
Cp1250	Windows Eastern Europe / Latin-2
Cp1251	Windows Cyrillic
Cp1252	Windows Western Europe / Latin-2
Cp866	PC Russian
MacThai	Macintosh Thai
UTF8	Standard UTF-8

```
bw.write(caratteri);
bw.flush();

// Legge il valore immesso da tastiera
br.read(caratteri, 0, lunghezza -1);
br.close();
bw.close();

// Visualizza il valore
lab_output.setText(new String(caratteri));
```

Osservandone i risultati si può notare come i caratteri accentati non sempre hanno una corrispondenza nel font del risultato. Per verificarlo basta provare una codifica di tipo `Cp1251` che corrisponde al Windows Cyrillic.

Infine si tenga presente che — poiché lo standard Unicode rappresenta ciascun carattere con due byte ovvero 16 bit, considerando però che molti caratteri che si utilizzano in un programma Java possono essere rappresentati attraverso un solo byte, e dato inoltre che molti calcolatori utilizzano una rappresentazione interna dei caratteri a otto bit — è stato reso disponibile un meccanismo per la rappresentazione dei caratteri Unicode in byte (anche per mantenere alta la portabilità del codice) attraverso i metodi

```
public final String readUTF() throws IOException
public static final String readUTF(DataInput in) throws IOException
della classe DataInputStream e
```

```
public final void writeUTF(String str) throws IOException
```

della DataOutputStream entrambe contenute nel package java.io.

I precedenti metodi permettono la lettura e scrittura dei caratteri di una stringa secondo il formato UTF-8.

Localizzazione: l'oggetto java.util.Locale

La classe Locale appartiene, come la classe ResourceBundle, al package java.util che contiene alcuni degli strumenti a supporto dell'internationalization. Il package java.text fornisce gli strumenti per la formattazione del testo, mentre in java.util sono stati aggiunti strumenti relativi alla localizzazione: infatti non solo è importante sapere come creare applicazioni portabili in funzione delle convenzioni geografiche ma è necessario anche poter sapere quale è il set di regole in uso, ovvero effettuare la localizzazione.

L'oggetto Locale serve proprio a questo, dato che, oltre a contenere le informazioni relative all'insieme di regole in uso entro un determinato ambito geografico/linguistico, permette di identificare l'ambito stesso.

In funzione dei parametri passati al costruttore è possibile creare un Locale funzione di una lingua, di una nazione o anche di alcuni parametri particolari.

Ad esempio il costruttore nella versione con il maggior numero di parametri ha la seguente firma

```
public Locale (String language, String country, String variant)
```

I valori dei parametri passati si riferiscono a precisi simboli definiti dallo standard ISO (International Standard Organization, l'organismo internazionale che cura a livello mondiale le regole di uniformazione).

Il primo di questi si riferisce al linguaggio ovvero alla lingua che ha predominanza nel particolare luogo, lingua che viene specificata tramite due lettere minuscole rappresentative della località e si riferisce allo standard ISO639. Per esempio la sigla per l'inglese in uso negli Stati Uniti è en mentre quella dell'italiano è it.

Va tenuto presente che non necessariamente in una stessa nazione si parla un'unica lingua. Per esempio esistono zone dell'Italia settentrionale in cui predominano il tedesco o il francese e ci sono nazioni come il Canada in cui, accanto a grandi aree di lingua inglese, esiste l'intero Quebec di lingua francese. Il secondo parametro del costruttore si riferisce proprio a questo: si tratta di una stringa costituita da caratteri questa volta maiuscoli (specificati nello standard ISO3166), che indicano la regione geografica: per la parte italiana della Svizzera ad esempio si utilizza la stringa "CH".

Per la definizione di un oggetto Locale basterebbero queste due proprietà, ma esiste anche la possibilità di fornire una variante ovvero permettere la specifica di strutture personalizzate implementate, ad esempio, da particolari tipi di browser. Per rappresentare un oggetto Locale si utilizza spesso la notazione del tipo language_country.

Ovviamente affinché l'oggetto sia dotato di una certa utilità sono stati introdotti nel JDK 1.1 altri oggetti *locale sensitive*, il cui comportamento cioè sia dipendente dal Locale impostato. Ognuno di questi oggetti dispone di metodi che permettono di specificare il Locale, oppure che utilizzano quello di default (si vedranno degli esempi in seguito). I metodi più importanti dell'oggetto Locale sono quindi:

```
public static synchronized void setDefault();
public static synchronized Locale getDefault();
```

Si tratta di metodi di classe e permettono rispettivamente di settare il Locale di default oppure di vedere quale sia quello corrente. Per eseguire delle operazioni *locale sensitive* si avranno allora due possibilità:

- utilizzare il Locale di default senza specificarlo nei metodi *locale sensitive*;
- utilizzare la versione dei metodi che hanno l'oggetto Locale come parametro.

Ultima considerazione riguarda la presenza di Locale predefiniti relativi ai Paesi più importanti, messi a disposizione attraverso variabili statiche definite nella stessa classe Locale. Per ottenere quello italiano basterà scrivere

```
Locale italia = Locale.ITALY;
```

Creazione di risorse localizzate: oggetto java.util.ResourceBundle

Come si è avuto modo di accennare, un meccanismo che permetta una semplice commutazione da un linguaggio a un altro deve essere accompagnato da un modo per conoscere la località di esecuzione: questo compito è svolto dall'oggetto ResourceBundle.

La filosofia di utilizzo di questo oggetto è molto simile a quella delle classi di supporto di un JavaBean. In questo caso insieme alla classe rappresentativa del componente, a supporto del componente possono esistere altre classi i cui nomi sono legati al nome della classe principale.

Per esempio, la classe che descrive esplicitamente il comportamento di un componente in termini di proprietà, eventi ed azioni, deve avere il nome MiaClasseBeanInfo dove MiaClasse è il nome della classe del bean.

Si supponga di essere nella necessità di scrivere un componente che debba essere venduto in tutto il mondo e quindi debba adattarsi a ogni ambito geografico/linguistico. Se il nome di tale componente è GlobalBean, allora sue proprietà potranno essere descritte nella classe GlobalBeanInfo: in questo caso per permettere l'internazionalizzazione sarebbe sufficiente fornire una classe con tale nome per ogni località.

Così facendo si dovrebbe creare una classe per ogni località e quindi compilare più classi diverse. Se poi volessimo aggiungere una nuova località a quelle disponibili si dovrebbe programmare un'altra classe e compilarla perdendo in eleganza e flessibilità.

Inoltre il compito di scegliere quale classe BeanInfo utilizzare in funzione del locale dovrebbe essere in questo caso a totale carico del programmatore e comunque prevista in fase di progettazione.

Sicuramente l'alternativa più semplice ed elegante potrebbe essere quella di creare un'unica classe GlobalBeanInfo e fornire ad essa un meccanismo più semplice per visualizzare le proprietà del bean in modo dipendente dalla località. Ad esempio si potrebbe pensare di creare un file di testo al quale la classe BeanInfo potrebbe accedere per reperire le informazioni di cui ha bisogno, ovvero i nomi di proprietà, eventi e metodi. Molto comodo sarebbe se la classe GlobalBeanInfo potesse accorgersi di questo file e ne utilizzasse autonomamente le informazioni.

La classe ResourceBundle permette di fornire le informazioni in modo *locale sensitive* attraverso gli stessi meccanismi con cui le classi BeanInfo le forniscono ai bean, sia estendendo tale classe oppure utilizzando semplici file testo che contengono i valori da utilizzare.

Altra analogia fra i bean e la internazionalizzazione è costituita dalla presenza di una regola per il nome che tali classi o file testo devono avere.

ResourceBundle: utilizzo e regole di naming

Per definire un insieme di risorse relative ad una particolare località, è possibile estendere la classe astratta ResourceBundle definendo i metodi:

```
protected abstract Object handleGetObject(String key) throws MissingResourceException

public abstract Enumeration getKeys()
```

Il primo ha come parametro un oggetto String che è il nome della risorsa cui si vuole accedere in modo *locale sensitive*. Il valore di ritorno sarà un Object rappresentativo della risorsa e adattato alla località. La sua assenza provoca una eccezione. Il metodo getKeys() invece ritorna l'insieme dei nomi delle risorse disponibili in quel ResourceBundle.

Tralasciando per il momento come questo sia possibile, si supponga di aver creato tanti oggetti ResourceBundle quante sono le località da gestire.

Primo passo da fare per accedere in modo *locale sensitive* a una risorsa è quella di ottenere l'oggetto ResourceBundle appropriato. Per fare questo si utilizzano le seguenti due versioni del metodo static getBundle():

```
public static final ResourceBundle getBundle(String baseName) throws MissingResourceException

public static final ResourceBundle getBundle(String baseName, Locale locale)
```

Anche in questo caso, come in molti metodi relativi all'internationalization, sono disponibili due versioni dello stesso metodo che differiscono nella specificazione o meno dell'oggetto Locale.

Se si utilizza la prima versione viene considerato come oggetto Locale quello di default eventualmente precedentemente settato con il metodo setDefault().

Si supponga adesso di disporre dell'oggetto ResourceBundle appropriato: per ottenere la risorsa relativa a una chiave particolare, basterà utilizzare il metodo getObject(), passando come parametro la chiave stessa: questo metodo ritornerà l'oggetto relativo alla chiave ed all'oggetto Locale utilizzato.

Al fine di evitare inutili operazioni di cast, dato che nella maggior parte dei casi gli oggetti sono delle stringhe, è fornito il metodo getString().

Per quanto riguarda le regole di naming è bene tener presente che ogni risorsa ha un nome, il cosiddetto baseName, che deve essere esplicativo della risorsa che vogliamo gestire in modo *locale sensitive*.

Si supponga di voler gestire le etichette di due pulsanti in due lingue diverse, ad esempio italiano ed inglese, relativamente ai messaggi "SI" e "NO". Il nome di questa risorsa potrebbe essere Bottone.

In base alle regole di naming, il nome della classe che estende ResourceBundle deve essere quello della risorsa a cui sono aggiunte, nel modo descritto, le stringhe identificatrici della lingua.

Quindi la classe che contiene le label dei bottoni italiani dovrà avere nome Bottone_it e la sua implementazione potrebbe essere

```
public abstract class Bottone_it extends ResourceBundle {
  public Object handleGetObject(String key) {
    if (key.equals("yes")) return "SI";
    else if (key.equals("no")) return "NO";
      return null;
  } // fine handleObject
}
```

Analogamente la versione inglese si chiamerà Bottone_uk e potrebbe essere implementata così:

```
public abstract class Bottone_uk extends ResourceBundle {
  public Object handleGetObject(String key) {
    if (key.equals("yes")) return "YES";
    else if (key.equals("no")) return "NO";
      return null;
  } // fine handleObject
}
```

Volendo creare un ResourceBundle per la parte italiana della Svizzera relativa a una minoranza indicata con il modificatore MI, il suo nome sarebbe stato Bottone_it_CH_MI.

Le classi create sono ancora astratte in quanto non è stato ridefinito il metodo getKeys() non utile al momento. Nel caso lo fosse, basterà creare un oggetto che implementi l'interfaccia java.util.Enumeration e ritornarlo come risultato. Il fatto che le classi create sono astratte non è comunque un problema in quanto non vengono mai istanziate direttamente con un costruttore, ma sono ottenute attraverso il metodo getBundle(). Se l'oggetto Locale di default è Locale_ITALY, si otterrà l'oggetto ResourceBundle attraverso il metodo:

```
ResourceBundle rb = ResourceBundle.getBundle("bottoni");
```

Mentre per ottenere quello inglese si dovrà esplicitare il Locale differente e si utilizzerà la seguente forma

```
ResourceBundle rb = ResourceBundle.getBundle("bottoni", new Locale("uk"));
```

L'utilizzo dei file di testo

In alternativa all'utilizzo di ResourceBundle si può ricorrere all'utilizzo di semplice file di testo in cui introdurre le varie versioni internazionalizzate di una stessa risorsa: ovviamente questa eventualità risulta utilizzabile solamente nel caso in cui tali risorse siano stringhe.

Si supponga ad esempio di voler specificare un Locale nel seugente modo

```
Locale locale= new Locale("lingua", "paese", "modif");
```

Se la risorsa si chiama bottoni, alla esecuzione del metodo getBundle(), nello spazio del ClassLoader inizia la ricerca di una classe di nome Bottoni_lingua_paese_modif. Se questa non è presente viene allora cercato un file di estensione properties dal nome

```
Bottoni_lingua_paese_modif.properties
```

Questo file che segue lo standard dei file Properties, avrà le varie coppie chiave–valore elencate ciascuna su una riga divise da un segno di uguale (=).

Nel caso in cui nemmeno tale file venga trovato, si generalizza la ricerca a un livello più alto, andando prima a cercare la classe denominata Bottoni_lingua_paese e successivamente il file Bottoni_lingua_paese.properties.

In caso negativo si procede a una ulteriore generalizzazione ricercando la classe Bottoni_lingua e il file Bottoni_lingua.properties. Infine sono cercati la classe Bottoni e il file Bottoni.properties.

Se nessuna di queste è stata trovata viene sollevata una eccezione del tipo MissingResourceException. È da notare che nel caso in cui venisse trovata una delle classi o file cercati, la ricerca non sarà interrotta. Questo perché esiste la possibilità di estendere un ResourceBundle da un altro e, ad esempio, fare in modo che la risorsa Bottoni_lingua_paese erediti delle informazioni da Bottoni_lingua. Questa è una caratteristica da non sottovalutare.

Le classi ListResourceBundle e PropertyResourceBundle

Si è accennato all'esistenza di classi del tipo ResourceBundle già presenti nel JDK 1.1. Queste classi, sempre del package java.util, sono ListResourceBundle e PropertyResourceBundle.

La ListResourceBundle è una classe astratta che fornisce le informazioni sotto forma di vettore di coppie di oggetti.

Ciascuno di questi oggetti è un vettore di due elementi: il primo è una stringa indentificatrice del dato, il secondo è il dato stesso. La classe è astratta e il suo utilizzo prevede la sua estensione e la definizione del metodo:

```
protected abstract Object[][] getContents();
```

Nel caso dell'esempio precedente il ListResourceBundle associato alla versione italiana sarebbe stato

```
// Versione italiana della ListResourceBundle
public class Bottone_it extends ListResourceBundle {
    protected Object[][] getContents() {
        return {{"yes","SI"},{"no","NO"}};
    }
}
```

La PropertyResourceBundle permette di ottenere le informazioni a partire da un InputStream. Questa classe non è astratta e dispone del costruttore:

```
public PropertyResourceBundle(InputStream in);
```

In particolare verranno cercate, nello stream, le coppie del tipo chiave=valore e da esse saranno ottenute le informazioni. Nel caso dell'esempio visto prima si sarebbe dovuto creare un file di nome inf.txt dato da

```
# file contenente le informazioni usate dal
# PropertyResourceBundle italiano
yes = SI
no = NO
```

e si sarebbe dovuto utilizzare il seguente codice

```
try {
    FileInputStream fis = new FileInputStream(new File("inf.txt"));
    PropertyResourceBundle prb = new PropertyResourceBundle(fis);
    String yes_it = prb.getString("yes");
    ...
    fis.close
}
catch(IOException ioe) {
    ...
}
```

In questo esempio è stato volontariamente messo un nome qualsiasi per il file che contiene le informazioni in quanto, se si utilizza esplicitamente PropertyResourceBundle, non è necessario che siano soddisfatte le regole di naming sopra descritte.

Le due classi appena viste sono utilizzate in modo completamente trasparente nella fase di ricerca a seguito della esecuzione del metodo getBundle().

A questo punto ci si potrebbe chiedere quando utilizzare un file di properties per specificare le grandezze *locale sensitive* e quando estendere la classe ResourceBundle.

Nel caso in cui le modifiche da fare siano relative solamente a del testo è bene utilizzare i file Properties che permettono di creare facilmente le varie versioni con un semplice text editor. Nel caso in cui le modifiche siano relative a oggetti non vi è altra scelta che creare le varie classi ResourceBundle ricordandosi, però, della possibilità di estendere una dall'altra e di ereditarne le informazioni.

La formattazione

Un aspetto molto importante legato all'internazionalizzazione di una applicazione è dato dalla gestione dei diversi formati di visualizzazione di stringhe particolari, come date, valute o altro.

La gestione delle date è forse uno dei problemi più frequenti quando ad esempio si devono manipolare dati memorizzati in tabelle di database, a causa dei differenti formati utilizzati. Il fenomeno legato all'anno 2000 ha insegnato molto da questo punto di vista.

Se relativamente alla gestione delle varie etichette della interfaccia grafica, la localizzazione permette di risolvere alcuni aspetti prettamente estetici, in questo caso l'utilizzo delle tecniche di internazionalizzazione permette di evitare l'insorgere di errori di funzionamento.

Per questo motivo la corretta formattazione può essere considerata un aspetto ancora più importante rispetto alla gestione delle risorse in modalità *locale sensitive*. Il primo aspetto che si può prendere in considerazione è dato dalla formattazione dei numeri

La formattazione dei numeri

La classe NumberFormat è molto utile in tal senso dato che permette di formattare numeri, valute e percentuali. Nel caso dei numeri è possibile formattare sia valori di tipo primitivo come double o int sia variabili reference, come Double o Integer.

Il metodo getNumberInstance() permette di ottenere un formattatore specifico in funzione del Locale impostato.

Ad esempio si potrebbe scrivere un piccolo programma NumberFormatter che formatta un numero in funzione del Locale specificato:

```
// due formattatori specifici
NumberFormat ItalianFormatter, EnglishFormatter;

// creazione dei Locale italiano e inglese americano
```

```
Locale ItalianLocale = new Locale("it", "IT");
Locale USLocale = new Locale("en", "US");

String EnglishTotalAmount, ItalianTotalAmount;

// variabile da formattare
Double TotalAmount = new Double (3425759.456);

// crea il formattatore italiano e formatta
ItalianFormatter = NumberFormat.getNumberInstance(ItalianLocale);
ItalianTotalAmount = ItalianFormatter.format(TotalAmount);
System.out.println("Formattazione " + Locale.ITALIAN + " :" + ItalianTotalAmount);

// crea il formattatore inglese americano e formatta
EnglishFormatter = NumberFormat.getNumberInstance(USLocale);
EnglishTotalAmount = EnglishFormatter.format(TotalAmount);
System.out.println("Formattazione " + Locale.ENGLISH + " :" + EnglishTotalAmount);
```

Tale programma produce il seguente risultato

```
Formattazione it : 3.425.759,456
Formattazione en : 3,425,759.456
```

La formattazione di valute in funzione del locale scelto avviene in maniera del tutto analoga, con la sola differenza che il metodo invocato per ottenere un formattatore è in questo caso il `getCurrencyInstance()`. Ad esempio, nel programma `CurrencyFormatter` si trova il seguente codice

```
Double TotalAmount = new Double (3425759.456);

NumberFormat ItalianFormatter, EnglishFormatter;

String EnglishTotalAmount, ItalianTotalAmount;

Locale ItalianLocale = new Locale("it", "IT");
ItalianFormatter = NumberFormat.getCurrencyInstance(ItalianLocale);
ItalianTotalAmount = ItalianFormatter.format(TotalAmount);
System.out.println("Somma in Euro: " + ItalianTotalAmount);

Locale USLocale = new Locale("en", "US");
EnglishFormatter = NumberFormat.getCurrencyInstance(USLocale);
EnglishTotalAmount = EnglishFormatter.format(TotalAmount);
```

```
System.out.println("Somma in Dollari: " + EnglishTotalAmount);
```

Il risultato dell'esecuzione in questo caso sarebbe

```
Somma in Euro: € 3.425.759
Somma in Dollari: $3,425,759.46
```

Ovviamente la trasformazione in funzione del Locale è valida solo da un punto di vista matematico, e non dall'effettivo cambio monetario delle valute utilizzate.

Meno importante in questo caso l'utilizzo di Locale differenti, anche se di fatto essi permettono di realizzare applicazioni più flessibili.

Formattazione personalizzata di cifre

Nel caso di valori corrispondenti a cifre decimali è possibile utilizzare le classi DecimalFormat e DecimalFormatSymbols sia per effettuare una formattazione personalizzata dei valori, indicando ad esempio il numero di zeri prefissi e postfissi, il carattere separatore delle migliaia e dei decimali.

In questo caso, al fine di personalizzare la formattazione di un valore decimale, la prima cosa da fare è definire il pattern di formattazione utilizzando le seguenti regole in formato BNF

```
pattern      :=  subpattern{;subpattern}
subpattern   :=  {prefix}integer{.fraction}{suffix}
prefix       :=  '\\u0000'..'\\uFFFD' - specialCharacters
suffix       :=  '\\u0000'..'\\uFFFD' - specialCharacters
integer      :=  '#'* '0'* '0'
fraction     :=  '0'* '#'*
```

dove le notazioni hanno il significato riportato nella tab. 3.2.

Tabella 3.2 – *Significato della notazione per la formattazione di un valore decimale.*

Notazione	Descrizione
X*	0 o più istanze di X
(X | Y)	X o Y
X..Y	un carattere qualsiasi da X a Y inclusi
S - T	tutti i caratteri in S eccetto quelli in T
{X}	X è opzionale

Tabella 3.3 – *Caratteri speciali per i sottopattern di formattazione di numeri decimali.*

Notazione	Descrizione
0	una cifra
#	una cifra, zero equivale ad assente
.	separatore decimale
,	separatore per gruppi di cifre
E	separa mantissa ed esponente nei formati esponenziali
;	separa i formati
-	prefisso per i negativi
%	moltiplica per 100 e mostra come percentuale
?	moltiplica per 1000 e mostra come "per mille"
{	segno di valuta; sostituito dal simbolo di valuta; se raddoppiato, sostituito dal simbolo internazionale di valuta; se presente in un pattern, il separatore decimale monetario viene usato
X	qualsiasi altro carattere può essere usato nel prefisso o nel suffisso
'	usato per indicare caratteri speciali in un prefisso o in un suffisso

Nello schema precedente il primo sottopattern è relativo ai numeri positivi, il secondo a quelli negativi. In ogni sottopattern è possibile specificare caratteri speciali che sono indicati nella tab. 3.3. Alcuni esempi di pattern utilizzati di frequente sono riportati nella tab. 3.4.

Un esempio potrebbe essere il seguente:

```
Locale ItalianLocale = new Locale("it", "IT");
NumberFormat NumberFormatter = NumberFormat.getNumberInstance(ItalianLocale);
DecimalFormat DecimalFormatter = (DecimalFormat) NumberFormatter;
DecimalFormatter.applyPattern(pattern);
String FormattedValue = DecimalFormatter.format(value);
System.out.println(pattern + " " + FormattedValue + " " + loc.toString());
```

che in funzione del pattern utilizzato e del locale impostato produce i risultati riportati in tab. 3.5.

Formattazione personalizzata

Grazie all'utilizzo della classe DecimalFormatSymbols è possibile modificare i vari simboli utilizzati come separatori: ad esempio è possibile specificare il carattere di separazione delle migliaia, dei decimali, il segno meno, quello di percentuale e quello di infinito.

Ad esempio

```
DecimalFormatSymbols MySymbols = new DecimalFormatSymbols(currentLocale);
unusualSymbols.setDecimalSeparator('-');
unusualSymbols.setGroupingSeparator('@');

String MyPattern = "#,##0.###";
DecimalFormat weirdFormatter = new DecimalFormat(MyPattern, MySymbols);
weirdFormatter.setGroupingSize(4);
```

Tabella 3.4 – *Pattern di utilizzo frequente.*

Value	Pattern	Output	Spiegazione
123456.789	###,###.###	123,456.789	Il segno del "diesis" (#) rappresenta una cifra, la virgola (,) è il separatore di gruppi di cifre e il punto (.) è il separatore decimale
123456.789	###.##	123456.79	Il value ha tre cifre a destra del punto decimale ma il pattern ne presenta solo due. Il metodo format risolve questa situazione con l'arrotondamento a due cifre.
123.78	000000.000	000123.780	Il pattern specifica degli zero prima e dopo i numeri significativi poiché, invece del diesis (#), viene usato il carattere 0
12345.67	$###,###.###	$12,345.67	Il primo carattere nel pattern è il segno del dollaro ($). Da notare che nell'output formattato, esso precede immediatamente la cifra più a sinistra.
12345.67	\u00A5###,###.###	¥12,345.67	Il pattern specifica il segno di valuta per lo yen giapponese (¥) con il valore Unicode 00A5.

Tabella 3.5 – *Risultato ottenuto in funzione di determinati pattern e locale.*

Pattern	Locale	Risultato
###,###.###	en_US	123,456.789
###,###.###	de_DE	123.456,789
### ###.###	fr_FR	123 456,789

```
String bizarre = weirdFormatter.format(235412.742);
System.out.println(bizarre);
```

L'esecuzione di questa porzione di codice produce il seguente risultato

```
2@35412-742
```

Formattazione di date e orari

Uno degli aspetti più importanti legati alla rappresentazioni di dati indipendentemente dalla zona geografica è dato dalla corretta gestione delle date e degli orari.

La classe DateFormat da questo punto di vista rappresenta un valido ausilio per tutte le operazioni di rappresentazione *locale independent* e di formattazione.

Al solito è possibile utilizzare formati standard oppure effettuare formattazioni sulla base di pattern particolari. Come prima cosa si prende in esame il caso della formattazione basata su formati predefiniti.

Formattazione predefinita

Per effettuare la formattazione di una data si deve per prima cosa creare un formattatore tramite il metodo getDateInstance() della classe DateFormat.

Successivamente, in modo del tutto analogo ai casi precedenti è necessario invocare il metodo format() per ottenere una rappresentazione testuale della data secondo il Locale utilizzato.

Ad esempio la parte centrale del programma DateFormatter potrebbe essere

```
// Locale italiano
Locale ItalianLocale = new Locale("it", "IT");
int Style = DateFormat.DEFAULT;
DateFormat ItalianDateFormatter;
ItalianDateFormatter = DateFormat.getDateInstance(Style, ItalianLocale);
String ItalianDate = ItalianDateFormatter.format(today);
System.out.println("Data formattata secondo lo standard italiano: " + ItalianDate);
```

Variando il Locale utilizzato, si otterrebbe il seguente risultato

```
Data formattata secondo lo standard italiano: 9-gen-04
Data formattata secondo lo standard USA: Jan 9, 2004
Data formattata secondo lo standard tedesco: 09.01.2004
```

Il metodo getDateInstance() viene fornito in tre versioni differenti

```
public static final DateFormat getDateInstance()
```

```
public static final DateFormat getDateInstance(int style)
public static final DateFormat getDateInstance(int style, Locale locale)
```

Mentre il parametro locale se specificato permette di ottenere un formattatore specializzato, il parametro style serve per specificare lo stile di visualizzazione della data: ad esempio, utilizzando il locale italiano e variando fra i valori DEFAULT, SHORT, MEDIUM, LONG e FULL, si possono ottenere i seguenti risultati

```
DEFAULT:  9-gen-04
SHORT:    09/01/04
MEDIUM:   9-gen-04
LONG:     9 gennaio 2004
FULL:     venerdì 9 gennaio 2004
```

La versione senza parametri di getDateInstance() restituisce un formattatore in base al Locale correntemente installato e con stile DEFAULT. Nel caso invece si desideri formattare orari, si dovrà ricavare un formattatore opportuno tramite il metodo getDateTimeInstance() disponibile anche in questo caso in quattro versioni differenti a seconda che si voglia specificare il Locale e lo stile:

```
public static final DateFormat getDateTimeInstance()
public static final DateFormat getDateTimeInstance(int dateStyle)
public static final DateFormat getDateTimeInstance(int dateStyle, int timeStyle)
public static final DateFormat getDateTimeInstance(int dateStyle, int timeStyle, Locale aLocale)
```

Rispetto a prima, adesso il programma TimeFormatter avrebbe di diverso solamente la parte di definzione del formattatore

```
DateFormat ItalianTimeFormatter;
ItalianTimeFormatter = DateFormat.getTimeInstance(DateFormat.DEFAULT, ItalianLocale);
```

Anche in questo caso la scelta dello stile di rappresentazione influisce sul risultato finale: variando fra DEFAULT e FULL si potranno ottenere i seguenti risultati

```
DEFAULT:  11.48.04
SHORT:    11.48
MEDIUM:   11.48.04
LONG:     11.48.04 GMT-07:00
FULL:     11.48.04 GMT-07:00
```

Se infine si desidera effettuare formattazioni di date e orari contemporaneamente si potrà ottenere il formattatore adatto grazie al metodo

```
public static final DateFormat getDateTimeInstance(int dateStyle, int timeStyle, Locale locale)
```

Il risultato che si ottiene dopo l'esecuzione utilizzando il Locale italiano e variando stile per la data e per l'orario è il seguente

```
DEFAULT:  9-gen-04 12.23.22
SHORT:    09/01/04 12.23
MEDIUM:   9-gen-04 12.23.22
LONG:     9 gen 2004 12.23.22 GMT-07:00
FULL:     venerdi 9 gennaio 2004 12.23.22 GMT-07:00
```

Formattazione personalizzata delle date

Facendo utilizzo della classe SimpleDateFormat si può definire un pattern di formattazione e personalizzare il formato voluto. Ecco di seguito un breve esempio

```
// Crea un oggetto con la data attuale
// da cui partire per la formattazione
Date now = new Date();

// prepara un formattatore per formattare una data
// nella forma 9/01/2004 12.09.59
String pattern = "dd/MM/yyyy H.mm.ss";
Locale ItLocale = new Locale("it","IT");
SimpleDateFormat DateTimeFormatter = new SimpleDateFormat(pattern, ItLocale);
String now_formatted = DateTimeFormatter.format(now);
```

Il funzionamento di questa classe è piuttosto intuitivo; per una completa rassegna dei caratteri utilizzabili e del loro significato per la composizione della stringa pattern si può far riferimento alla tab. 3.6. Ogni carattere ad eccezione di quelli contenuti in ['a'..'z'] e ['A'..'Z'] viene utilizzato come separatore. Ad esempio i caratteri : . [spazio] , # @ appariranno anche se non racchiusi fra apici. Ad esempio i seguenti pattern producono i risultati riportati di seguito

```
"yyyy.MM.dd G 'at' hh:mm:ss z"      —>   2004.01.09 AD at 15:08:56 PDT
"EEE, MMM d, ''yy"                  —>   Wed, Jan 9, '04
"h:mm a"                            —>   12:08 PM
"hh 'o''clock' a, zzzz"             —>   12 o'clock PM, Pacific Daylight Time
"K:mm a, z"                         —>   0:00 PM, PST
"yyyyy.MMMMM.dd GGG hh:mm aaa"      —>   2004.January.9 AD 12:08 PM
```

Il programma contenuto nella classe SimpleDateFormat contiene inoltre un interessante e utile metodo, parse(), il quale è in grado di eseguire la traduzione inversa da una stringa rappresentante una data a oggetto Date. In questo caso si tenga presente che la mancanza di informazione derivante dall'utilizzo di sole due cifre per l'anno viene risolta in maniera drastica: viene infatti

Tabella 3.6 – *Significato dei caratteri utilizzabili nella stringa pattern della classe SimpleDateFormat.*

simbolo	significato	risultato	esempio
G	era	testo	AD
y	anno	numerico	1996
M	mese dell'anno	testo & numerico	luglio & 07
d	giorno del mese	numerico	10
h	ora in am/pm (1~12)	numerico	12
H	ora del giorno(0~23)	numerico	0
m	minuto dell'ora	numerico	30
s	secondi del minuto	numerico	55
S	millisecondi	numerico	978
E	giorno della settimana	testo	giovedì
D	giorno dell'anno	numerico	189
F	giorno della settimana	numerico	2 (2° mer di luglio)
w	settimana dell'anno	numerico	27
W	settimana del mese	numerico	2
a	marcatore am/pm	testo	PM
k	ora del giorno (1~24)	numerico	24
K	ora in am/pm (0~11)	numerico	0
z	timezone	testo	Pacific Standard Time
'	escape per testo	delimitatore	
''	apice singolo	carattere	'

effettuato un aggiustamento della data considerando secolo corrente quello compreso dagli 80 anni prima ai 20 dopo la data di creazione dell'istanza della SimpleDateFormat. In base a tale regola, utilizzando un pattern MM/dd/yy ed un SimpleDateFormat creato il 12 gennaio 1999, gli estremi per considerare il secolo saranno 1919 e 2018. Quindi il numero 64 verrà considerato come 1964, mentre il 17 come 2017, come riportato in fig. 3.2.

Figura 3.2 – *Nel caso in cui l'anno sia espresso con due sole cifre, il metodo parse effettua un'approssimazione.*



Gestione dei messaggi

Si è visto finora come modificare contenuto o formattazione di una stringa in funzione del Locale impostato. Anche se la flessibilità ottenibile con tali tecniche è sicuramente alta, si tratta pur sempre di utilizzare archivi di messaggi prestabiliti e non modificabili.

Si consideri ad esempio la gestione di messaggi di errore che devono essere visualizzati in apposite finestre di dialogo. In questo caso particolare infatti il testo che deve essere visualizzato spesso non può essere prestabilito a priori, essendo dipendente dal caso specifico.

Supponendo di dover visualizzare un messaggio indicante il numero di messaggi di posta elettronica presenti in una determinata casella, si potrebbe pensare di preparare i seguenti messaggi

```
"Nella casella " + PopAccount + " ci sono " + NumMessages + " messaggi"
```

oppure

```
"Nella casella" + PopAccount + " non ci sono messaggi"
```

Queste due semplici opzioni non garantiscono sufficiente flessibilità, se si pensa al caso in cui sia presente un solo messaggio sul server, quando si dovrebbe ricorrere ad uno schema del tipo

```
"Nella casella " + PopAccount + " vi è un solo messaggio"
```

Inoltre non tutte le lingue utilizzano lo stesso schema soggetto–verbo–complemento per cui a volte non è sufficiente rimpiazzare le parole che cambiano al variare della lingua, ma si deve modificare tutta la struttura della frase.

Una soluzione potrebbe essere quella di aggiungere un numero sufficiente di template in modo da coprire tutte le possibili casistiche: questa strada, oltre a non essere particolarmente elegante, complica le cose nel caso in cui, oltre alla gestione del singolare/plurale, si deve tener conto anche del genere (maschile/femminile).

In definitiva la gestione di messaggi di vario tipo deve tener conto della gestione delle frasi composte e del plurale.

Messaggi composti

Riprendendo in esame il caso precedente, si immagini di voler gestire una frase del tipo

Alle ore <u>12.23</u> del giorno <u>09/01/04</u> nella casella x34f erano presenti <u>33</u> messaggi.

dove le parole sottolineate rappresentano le parti variabili della frase. Per il momento si tralascia il problema del singolare della parola "messaggi".

Successivamente all'individuazione delle parti variabili del messaggio si deve creare un pattern che corrisponda alla frase da gestire, pattern che potrà essere memorizzato in un file di proprietà; ad esempio tale file potrebbe essere così strutturato

```
template = Alle ore {2,time,short} del giorno
{2,date, long} nella casella {0} erano presenti {1,number,integer} messaggi.
mailbox = x34f
```

la prima riga di tale file contiene il pattern generico da utilizzare per ottenere la frase finale. Le parentesi graffe contengono le parti variabili che dovranno essere sostituite di volta in volta, e il loro significato è riportato nella tab. 3.7.

È necessario creare quindi un array di oggetti dove memorizzare le varie parti da sostituire nel pattern.

```
Object[] messageArguments = {
   messages.getString("planet"),
   new Integer(7),
   new Date()
};
```

Ovviamente si potrebbe pensare di rendere le cose ancora più eleganti e automatizzate creando un file di risorse anche per queste variabili.

Il passo successivo consiste nel creare un formattatore di messaggi, ovvero

```
MessageFormat formatter = new MessageFormat("");
formatter.setLocale(ItalianLocale);
```

sul quale si applica il Locale corrente per permettere la corretta visualizzazione di date e simili.

Tabella 3.7 – *Argomenti per template in MessageBundle_en_US.properties.*

Argomento	Descrizione
{2,time,short}	La porzione relativa al tempo dell'oggetto Date. Lo stile short specifica lo stile di formattazione DateFormatSHORT.
{2,date,long}	La porzione relativa alla data dell'oggetto Date. Lo stesso oggetto Date viene usato per le variabili sia di data che di tempo. Nell'array di argomenti di Object, l'indice dell'elemento che detiene l'oggetto Date è 2 (vedere successiva descrizione).
{1,number,integer}	Un oggetto number ulteriormente specificato con lo stile numerico integer.
{0}	La String nel ResourceBundle corrispondente alla chiave planet.

Infine si deve provvedere ad applicare il formattatore al pattern, sostituendo i parametri variabili con quelli presenti nell'array di oggetti appena visto:

```
formatter.applyPattern(MailMessages.getString("template"));
String msg = formatter.format(MessagesArguments);
```

Il risultato che si ottiene nel caso in cui il locale sia quello italiano

```
Alle ore 13.42 del giorno 9 gennaio 2004 nella casella
X34f erano presenti 33 messaggi.
```

Mentre per quello inglese statunitense

```
At 1:41 PM on January 9, 2004 the message box X34f contains 33 messages.
```

Confronto fra caratteri

Molto di frequente può accadere di dover controllare se un certo carattere corrisponda a una cifra oppure a una lettera: si pensi ad esempio a tutti i controlli che spesso sono necessari sui dati immessi dall'utente tramite un form. In questo caso la soluzione che tipicamente si adotta è quella di confrontare il valore del codice (ASCII o più propriamente Unicode) dei caratteri immessi: ad esempio si potrebbe scrivere

```
char c;

if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'))
   // c  è una lettera

if (c >= '0' && c <= '9')
   // c  è un numero

if ((c == ' ') || (c == '\n') || (c == '\t'))
   // c  è una carattere speciale
```

Benché questa sia la soluzione tipicamente utilizzata nella maggior parte dei casi, è errata, perché effettua un semplice controllo sull'ordinamento fornito dai codici Unicode dei caratteri, ma considera le regole grammaticali presenti nelle diverse lingue. Per ovviare a questo problema si possono utilizzare i vari metodi messi a disposizione dalla classe `Character`, quali

```
public static boolean isDigit(char ch)
public static boolean isLetter(char ch)
public static boolean isLetterOrDigit(char ch)
```

```
public static boolean isLowerCase(char ch)
public static boolean isUpperCase(char ch)
```

Usando tali metodi il controllo sui caratteri può essere effettuato ad esempio come segue:

```
char c;
if (Character.isLetter(ch))
  ...
if (Character.isDigit(ch))
  ...
if (Character.isSpaceChar(ch))
  ...
```

Inoltre il metodo getType() permette di ricavare la categoria di un carattere: nel caso di caratteri dell'alfabeto è possibile ricavare se si tratta di una lettera maiuscola o minuscola, oppure nel caso di caratteri speciali se si tratta di un simbolo matematico o di un simbolo di punteggiatura.

Un approfondimento su queste indicazioni è presente nella documentazione ufficiale Sun.

Ordinamento

Oltre a voler conoscere la categoria di un carattere, un altro tipo di operazione molto utile è determinare l'ordinamento dei caratteri, al fine ad esempio di elencare in ordine alfabetico una serie di parole. Per ottenere risultati corretti, anche in questo caso è indispensabile tenere conto delle regole grammaticali della lingua e zona geografica scelte.

Infatti, anche se può apparire ovvio che il carattere "a" sia sempre minore del carattere "b", si possono avere situazioni meno ovvie, come il caso delle lettere accentate o di lettere doppie. Inoltre in alcuni rari casi potrebbe essere necessario specificare un nuovo set di regole di precedenza, senza dover per forza rispettare quelle del Locale in uso. La classe Collator a tal proposito permette di effettuare gli ordinamenti sulla base delle regole grammaticali della lingua utilizzata. Per ottenere un Collator si può ad esempio scrivere

```
Collator DefaultCollator = Collator.getInstance();
```

se si vuole utilizzare la lingua del Locale di default, oppure

```
Collator ItalianCollator = Collator.getInstance(Locale.ITALIAN);
```

nel caso si voglia specificare il Locale da utilizzare. Come si può notare in questo caso è sufficiente specificare la lingua e non anche il Paese, dato che si assume che le regole di ordinamento non dipendono dalla localizzazione geografica, ma esclusivamente dalla grammatica.

Una volta in possesso di un Collator non è necessario indicare nessun altro parametro, dato che le regole grammaticali sono contenute nella classe, e si potrà quindi utilizzare il metodo compare per effettuare le comparazioni fra caratteri stringhe o semplicemente fra caratteri. Ad esempio

```
ItalianCollator.compare("aaa", "aab")
```

restituirà –1 visto che nella nostra lingua la stringa "aaa" è minore rispetto ad "aab".

Utilizzando due Collator differenti, uno per l'Italiano e l'altro per il Francese, si può provare a effettuare un ordinamento sul set di parole

```
{perché, péché, è, pêche, pesce}
```

Nel primo caso si ottiene

```
è, péché, pêche, perché, pesce
```

Mentre per la lingua francese il risultato è

```
è, pêche, péché, perché, pesce
```

Nel caso in cui si desideri utilizzare un set di regole di ordinamento personalizzate si può ricorrere alla classe RuleBasedCollator, specificando le regole di ordinamento per mezzo di una ótringa che viene poi passata al costruttore RuleBasedCollator. Ad esempio si potrebbe scrivere

```
String MyRule = "< a < b < c < d";
RuleBasedCollator MyCollator = new RuleBasedCollator(MyRule);
```

In questo caso il Collator creato effettuare le comparazioni utilizzando come unica regola base il fatto che la lettera "a" è minore di "b" che a sua volta è minore di "c" e di "d".

Ovviamente affinché abbia senso utilizzare un set di regole personalizzate, è necessario specificare un ordinamento completo su tutte le lettere dell'alfabeto e simboli vari che si suppone saranno utilizzate nel programma. Prendendo ad esempio il caso riportato nella documentazione della Sun, si potrebbe pensare di creare due set di regole di ordinamento, uno per la lingua inglese e uno per la lingua spagnola.

```
// ordinamento lingua inglese
String enRules = ("< a,A < b,B < c,C < d,D < e,E < f,F "
                + "< g,G < h,H < i,I < j,J < k,K < l,L "
                + "< m,M < n,N < o,O < p,P < q,Q < r,R "
                + "< s,S < t,T < u,U < v,V < w,W < x,X "
                + "< y,Y < z,Z");

// ordinamento lingua spagnola
String smallnTilde = new String("\u00F1"); // ñ
String capitalNTilde = new String("\u00D1"); // Ñ

String spRules = ("< a,A < b,B < c,C "
```

```
+ "< ch, cH, Ch, CH "
+ "< d,D < e,E < f,F "
+ "< g,G < h,H < i,I < j,J < k,K < l,L "
+ "< ll, lL, Ll, LL "
+ "< m,M < n,N "
+ "< " + smallnTilde + "," + capitalNTilde + " "
+ "< o,O < p,P < q,Q < r,R "
+ "< s,S < t,T < u,U < v,V < w,W < x,X "
+ "< y,Y < z,Z");
```

Si noti come in questo caso siano stati introdotti i caratteri speciali "ñ" e "Ñ" e come la doppia "ll" sia stata forzatamente inserita dopo la singola "l", cosa che rende vera la relazione

```
"lu" < "ll"
```

Il programma principale potrebbe quindi effettuare l'ordinamento utilizzando i due Locale inglese e spagnolo, come ad esempio

```
try {
   RuleBasedCollator enCollator = new RuleBasedCollator(enRules);
   RuleBasedCollator spCollator = new RuleBasedCollator(spRules);

   sortStrings(enCollator, words);
   printStrings(words);

   System.out.println();

   sortStrings(spCollator, words);
   printStrings(words);
}
catch (ParseException pe) {
   System.out.println("Parse exception for rules");
}
```

dove words è un array di stringhe.

Migliorare l'efficienza delle operazioni di ordinamento

L'operazione di comparazione può risultare costosa da un punto di vista computazionale, per cui, quando si debbano svolgere frequentemente ordinamenti sullo stesso insieme di parole, può essere conveniente utilizzare una tecnica alternativa, basata sull'oggetto CollationKey.

In questo caso si genera una chiave per ogni parola da confrontare, e si provvede a effettuare successivamente la comparazione sulle chiavi, piuttosto che direttamente sulle parole.

Dato che la generazione delle chiavi è una operazione costosa, risulta conveniente utilizzare questa tecnica solo nel caso in cui si debba effettuare più volte la comparazione sullo stesso insieme di parole, in modo da ammortizzare il tempo iniziale necessario per la produzione delle chiavi.

Qui di seguito sono riportati alcuni passi del programma completo di ordinamento basato su chiavi:

```
Locale ItalianLocale = new Locale ("it", IT");
Collator ItalianCollator = Collator.getInstance (ItalianLocale);
CollationKey[] keys = new CollationKey[words.length];

for (int k = 0; k < keys.length; k ++)
   keys[k] = ItalianCollator.getCollationKey(words[k]);

// ordina gli elementi dell’array delle chiavi
CollationKey tmp;
for (int i = 0; i < keys.length; i++) {
   for (int j = i + 1; j < keys.length; j++) {
      if (keys[i].compareTo(keys[j]) > 0) {
         tmp = keys[i];
         keys[i] = keys[j];
         keys[j] = tmp;
      }
   }
}
```

dove al solito `words` è un array di stringhe. Una volta che l’array delle chiavi è ordinato, si può risalire alle stringhe originarie per mezzo del metodo `getResourceString()`, come ad esempio

```
for (int i = 0; i < keys.length; i++) {
   System.out.println(keys[i].getSourceString());
}
```

La classe BreakIterator

Dato un testo espresso in una determinata lingua, il processo di separazione in caratteri, parole, linee o frasi è sicuramente dipendente dalle regole grammaticali in vigore in tale lingua.

Va considerato che in certe lingue alcune parole sono espresse da un solo carattere del particolare alfabeto, in altre lingue si scrive da destra verso sinistra, in alcune si trascrivono le sole consonanti.

Ad esempio, la parola araba طريق , ṭarīq , che significa “strada”, può essere rappresentata dai caratteri Unicode

```
String street = "\ufed6" + "\ufef3" + "\u0631" + "\ufec1";
```

Questo semplice esempio fa comprendere come in realtà l'individuazione dei caratteri non sia un compito semplice.

La classe BreakIterator, per mezzo dei metodi

```
public static BreakIterator getCharacterInstance()
public static BreakIterator getCharacterInstance(Locale where)
public static BreakIterator getWordInstance()
public static BreakIterator getWordInstance(Locale where)
public static BreakIterator getSentenceInstance()
public static BreakIterator getSentenceInstance(Locale where)
public static BreakIterator getLineInstance()
public static BreakIterator getLineInstance(Locale where)
```

permette di ottenere delle istanze specifiche in funzione del tipo di separatore che si intende ottenere.

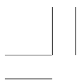
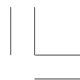
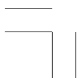
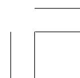
Il tipo di BreakIterator più semplice è quello che permette di individuare i singoli caratteri all'interno di una stringa. Eccone un semplice esempio

```
Locale ItLocale = new Locale ("it", IT");
BreakIterator ItalianCharIterator;
ItCharIterator = BreakIterator.getCharacterInstance(ItLocale);
ItCharIterator.setText(StringToBreak);
int CharPosition = ItCharIterator.first();

while (CharPosition != BreakIterator.DONE) {
  System.out.println (CharPosition);
    CharPosition = ItCharIterator.next();
}
```

dove StringToBreak rappresenta una stringa da suddividere in caratteri, e che, nel caso della parola araba di cui prima, restituirebbe la sequenza 0,2,4,6.

L'utilizzo degli altri metodi è similare e molto intuitivo, per cui si rimanda agli esempi allegati per maggiori approfondimenti o alla documentazione ufficiale della Sun.

Capitolo 4

Java e XML

ANDREA GIOVANNINI

Introduzione

XML è una tecnologia avviata a un utilizzo pervasivo nel Web Publishing e nelle applicazioni *business to business*. Java è il linguaggio ideale per questo tipo di applicazioni, grazie alle sue caratteristiche di portabilità ed estendibilità, ed è quindi naturale che le due tecnologie si incontrino. In questo capitolo si vedranno i concetti principali di XML e si approfondiranno alcuni temi specifici della programmazione di applicazioni XML con Java.

Fondamenti di XML

Si affronteranno ora gli aspetti principali di questa tecnologia, riprendendo poi con qualche esempio di programmazione Java i concetti presentati. Questa sezione non vuole comunque essere un tutorial completo di XML, ma una semplice introduzione agli aspetti più avanzati della tecnologia che verranno affrontati nel prosieguo del capitolo.

Che cosa è XML

XML (eXtensible Markup Language) [1] è un metalinguaggio per definire nuovi linguaggi basati su tag. Per questo aspetto XML è molto simile a HTML, ma c'è una differenza fondamentale: XML è una notazione per definire linguaggi mentre HTML è un particolare linguaggio. I punti chiave di XML sono i seguenti:

- Separazione del contenuto dalla rappresentazione dei dati: un documento XML definisce la struttura dei dati e non contiene alcun dettaglio relativo alla loro formattazione o a un qualsiasi utilizzo.

- Definizione di un formato standard: XML è uno standard del W3C e quindi un documento XML può essere elaborato da qualsiasi parser o tool conforme allo standard.

Segue ora un semplice esempio di file XML:

```
<?xml version="1.0"?>
<todolist>
  <item>
    <number>1</number>
    <priority>6</priority>
    <description>Leggere la posta</description>
    <state>2</state>
  </item>
  <item>
    <number>2</number>
    <priority>9</priority>
    <description>Riunione</description>
    <state>2</state>
  </item>
  <item>
    <number>3</number>
    <priority>8</priority>
    <description>Andare a correre nel parco</description>
    <state>1</state>
  </item>
</todolist>
```

Il documento precedente definisce una *todo-list*, ovvero un elenco di attività e di informazioni ad esse associate come la priorità e lo stato di avanzamento. Il primo elemento del documento è l'intestazione tipica di ogni file XML

```
<?xml version="1.0"?>
```

Quindi il documento presenta le informazioni relative alla todo-list organizzate in una gerarchia di elementi; i tag utilizzati sono funzionali alla struttura del documento e non contengono alcuna informazione relativa all'utilizzo della todo-list. Ad esempio dal documento è chiaro che l'attività `Riunione` ha una priorità maggiore rispetto a `Leggere la posta` ma non è possibile evincere nulla relativamente al modo in cui la todo-list possa essere visualizzata e in che tipo di documento.

Struttura di un documento XML

Così come un documento HTML deve essere redatto secondo alcune regole sintattiche ben precise, affinché sia visualizzabile da un browser, anche un documento XML deve rispettare ben precise regole strutturali; in particolare

- ogni tag aperto deve essere chiuso;

- i tag non devono essere sovrapposti;
- i valori degli attributi devono essere racchiusi fra "";
- i caratteri < > e " " nel testo di un file XML devono essere rappresentati dai caratteri speciali &lt &gt e &quot.

I documenti XML che rispettano le regole precedenti vengono detti ben formati (*well-formed*). Oltre a poter definire documenti ben formati è possibile specificare la particolare struttura di un file XML, ad esempio per garantire che il tag description sia compreso all'interno del tag item. In una sezione successiva si approfondiranno queste problematiche.

Elaborazione di documenti XML

Nei paragrafi precedenti sono stati illustrati alcuni aspetti fondamentali di XML. A questo punto si devono mettere in pratica i concetti visti cominciando a parlare di elaborazione di documenti XML.

Dato un documento XML, questo sarà ragionevolmente elaborato da un'applicazione per vari scopi (eseguire calcoli sui dati contenuti, visualizzarli in una pagina web, trasmetterli a un altro programma, …). Alla base di tutto ci sarà quindi un *parser* XML ovvero un opportuno strumento per assicurare che il documento sia ben formato e valido. Sorge ora il problema di come usare i dati contenuti all'interno del documento; non è infatti pensabile di ragionare in termini di elaborazione di stringhe ed è quindi necessaria una API che permetta di elaborare il documento a un livello di astrazione più elevato.

Le sezioni successive faranno uso di JAXP, *Java API for XML Processing*, una API sviluppata da Sun per incapsulare l'utilizzo di un particolare parser XML inclusa nel JDK 1.4. Oltre a JAXP sono state realizzate altre API:

- Java API for XML/Java Binding (JAXB): libreria per la gestione del data binding fra documenti XML e oggetti Java (verrà approfondita in una sezione successiva);
- Long Term JavaBeans™ Persistence;
- Java API for XML Messaging (JAXM): messaging con documenti XML;
- Java API for XML RPC (JAX-RPC);
- Java API for XML Registry (JAXR).

Per maggiori informazioni su JAXP e le varie API sviluppate da Sun si rimanda a [3]. Per gli esempi proposti, il parser di riferimento è Xerces [18], sviluppato all'interno dell'Apache XML Project (http://xml.apache.org).

SAX

Architettura

L'architettura di SAX (Simple API for XML) è molto semplice ed è basata su un meccanismo a eventi: il parser legge il documento XML e lancia una serie di eventi (inizio documento, inizio elemento, ..., fine documento) che vengono intercettati da un'oggetto che implementa una particolare interfaccia. SAX sfrutta quindi il pattern Observer per disaccoppiare la lettura del documento XML dall'elaborazione client dello stesso; per questo motivo vengono definite due interfacce: XMLReader che rappresenta il parser e ContentHandler che intercetta le azioni del parser, entrambe definite nel package org.xml.sax.

L'interfaccia XMLReader è implementata da chi ha realizzato il parser, mentre chi deve elaborare il documento XML implementerà ContentHandler definendo i metodi di callback che vengono invocati in corrispondenza degli eventi.

L'interfaccia ContentHandler

Si vedranno ora in dettaglio i metodi principali dell'interfaccia ContentHandler:

```
package org.xml.sax;

public interface ContentHandler {
   public void setDocumentLocator(Locator locator);
   public void startDocument() throws SAXException;
   public void endDocument() throws SAXException;
   public void startPrefixMapping(String prexif, String uri) throws SAXException;
   public void endPrefixMapping(String prexif) throws SAXException;
   public void startElement(String namespaceURI, String localName, String qName, Attributes atts)
                throws SAXException;
   public void endElement(String namespaceURI, String localName, String qName) throws SAXException;
   public void characters(char[] ch, int start, int length) throws SAXException;
   public void ignorableWhitespace( char[] ch, int start, int length ) throws SAXException;
   public void processingInstruction(String target, String data) throws SAXException;
   public void skippedEntity(String name) throws SAXException;
}
```

Il metodo setDocumentLocator() imposta un oggetto Locator che permette di risalire al punto del documento che ha generato un evento di parsing.

L'inizio e la fine del parsing di un documento vengono intercettati dai metodi startDocument() ed endDocument() utilizzati ripettivamente per inizializzare l'handler, se necessario, e per eseguire particolari azioni al termine dell'output, come clean-up o generazione di un particolare output per comunicare l'esito del parsing.

Il metodo endDocument() non viene comunque richiamato se il parser ha incontrato un errore: in tal caso infatti il parsing viene interrotto e si generano opportuni eventi di gestione degli errori.

I metodi startPrefixMapping() ed endPrefixMapping() intercettano le associazioni fra URI e namespace, come ad esempio xmlns:moka="http://www.mokabyte.it/2003".

I metodi più interessanti di ContentHandler sono quelli di gestione dei tag, startElement() ed endElement(), chiamati rispettivamente quando il parser incontra un tag aperto (<FOO>) e un tag chiuso (</FOO>). Per poter elaborare le informazioni contenute nel tag, il metodo startElement() ha come parametri

- l'URI associato al namespace;
- il nome locale dell'elemento;
- il nome qualificato dell'elemento;
- gli attributi associati al tag rappresentati come un oggetto Attributes.

La firma del metodo startElement() è la seguente:

```
public void startElement(String URI, String lName, String qName, Attributes attr)
throws SAXException;
```

I parametri permettono di identificare i vari componenti di un elemento, comprese le informazioni sui namespace. Si supponga ad esempio di avere il seguente tag

```
<list:item xmlns:list="http://www.foo.com/ns">
```

In questo caso il namespace URI è http://www.foo.com/ns mentre il nome locale è item. Se il parser è configurato per l'elaborazione dei prefissi, allora i metodi startElement() e endElement() riportano list:item come nome qualificato altrimenti tale parametro potrebbe non essere valorizzato. La sequenza degli eventi generati è startPrefixMapping(), startElement() quindi gli eventi relativi a eventuali elementi annidati e infine endElement() ed endPrefixMapping().

Si osservi che i tag di un documento XML possono essere arbitrariamente annidati ma il metodo startElement() non comunica alcuna informazione relativa al contesto in cui un elemento viene incontrato. È quindi onere del programmatore tenere traccia delle sequenze dei tag, usando ad esempio uno stack.

Il metodo characters() viene invocato per comunicare all'handler il contenuto del documento compreso fra due tag. I dati vengono passati come un array di byte: sarà il programmatore a convertirlo eventualmente in una stringa. Nel caso in cui un elemento contenga spazi bianchi sarà il metodo ignorableWhitespace() a processarli.

Se il documento esaminato contiene delle processing instruction allora il parser invoca il metodo processingInstruction() sull'handler.

Nel caso in cui un parser non validante incontri un riferimento ad un'entità (p.e. &doc) non ha modo di recuperare il testo da sostituire o di verificare che si tratti di un simbolo speciale (p.e.

è); per risolvere il dilemma viene quindi invocato il metodo skippedEntity() passando come parametro il testo incriminato.

Per semplificare il compito dei programmatori, viene fornita la classe DefaultHandler, definita nel package org.xml.sax.helpers, che implementa tutti i metodi di ContentHandler come no-operation; i metodi sono cioè vuoti. In questo modo per definire un proprio handler è sufficiente estendere DefaultHandler e ridefinire solo i metodi necessari.

Si consideri ora il documento XML visto in precedenza

```
<?xml version="1.0"?>
<todolist>
  <item>
    <number>1</number>
    <priority>6</priority>
    <description>Leggere la posta</description>
    <state>2</state>
  </item>
  . . .
</todolist>
```

La tab. 4.1 mostra la corrispondenza fra gli elementi del documento e gli eventi SAX generati

Dato il documento XML precedente si supponga di voler contare il numero di attività: la seguente classe ItemCounter esegue questo compito

```
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class ItemCounter extends DefaultHandler {
  private int counter = 0;

public void startElement(String uri, String name, String qName, Attributes attributes) throws SAXException {
```

Tabella 4.1 – *Corrispondenza tra elementi di un documento ed eventi SAX generati.*

Elementi	Eventi SAX
<todolist>	startDocument()
<item>	startElement()
<number>	startElement()
1	characters()
</number>	endElement()
<priority>	startElement()
6	characters()
</priority>	endElement()
...	...
</item>	endElement()
...	...
</todolist>	endDocument()

```
        if (qName.equals("item")) {
          counter++;
        }
      }

      public void endDocument() {
        System.out.println("Trovate " + counter + " attività");
      }
    }
```

Nella sezione relativa alla validazione dei documenti si vedrà anche come gestire gli errori di parsing.

Parsing

Il parsing di un documento XML viene eseguito da un oggetto che implementa XMLReader; si potrebbe quindi specificare la classe del particolare parser (Xerces, Crimson, …) ma per incapsulare questo processo di creazione e renderlo parametrico viene fornita la classe factory org.xml.sax.helpers.XMLReaderFactory; il seguente frammento di codice mostra il suo utilizzo:

```
import org.xml.sax.*;
import org.xml.sax.helpers.*;

...
  XMLReader parser = XMLReaderFactory.createXMLReader();
  ContentHandler handler = new ItemCounter();
  parser.setContentHandler(handler);
  parser.setErrorHandler(new MyErrorHandler());
  parser.parse( new InputSource(args[0]) );
...
```

Con XMLReaderFactory.createXMLReader() si ottiene un'istanza del parser la cui classe specifica è definita nella property di sistema org.xml.sax.driver. Quindi si impostano gli handler precedentemente definiti e il parser può ora "macinare" documenti XML. Per incapsulare i possibili input del parser, si utilizza la classe org.xml.sax.InputSource; in questo caso viene specificato un URI a un file.

SAX e JAXP

Il supporto di JAXP per la API SAX è dato dalle seguenti classi:

- SAXParserFactory: classe factory che crea l'istanza del parser specificata dalla property javax.xml.parsers.SAXParserFactory;

- SAXParser: incapsula l'implementazione del parser referenziando un XMLReader; è possibile accedervi mediante il metodo getXMLReader().

È possibile riscrivere il codice precedente nel modo seguente:

```
import javax.xml.parsers.*;

  ...
  SAXParserFactory spf = SAXParserFactory.newInstance();
  SAXParser parser = spf.newSAXParser();
  parser.setContentHandler(handler);
  parser.setErrorHandler(new MyErrorHandler());
  parser.parse( new InputSource(args[0]) );
  ...
```

DOM

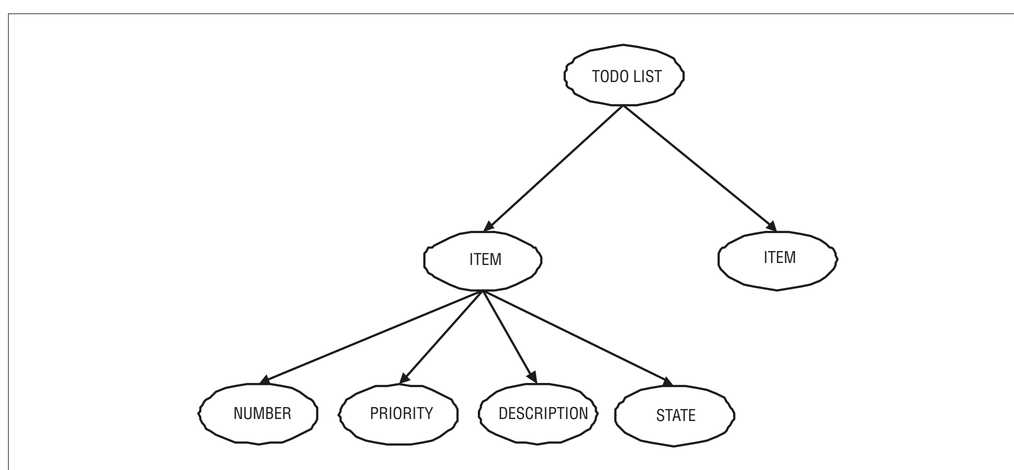
Introduzione

DOM è un modello a oggetti standard per la gestione di documenti XML, HTML e, in prospettiva, WML. La principale differenza rispetto a un parser SAX consiste nel fatto che, come visto, l'elaborazione in SAX è sequenziale e avviene nel corso del parsing. Le informazioni sul documento non sono più disponibili al termine dell'elaborazione a meno che il programmatore non si sia costruito una struttura ausiliaria. DOM svolge proprio questa funzione, ovvero esegue il parsing del documento XML e restituisce una struttura gerarchica ad albero che lo rappresenta; dopo l'esecuzione del parsing è quindi possibile elaborare più volte la stessa struttura.

Il W3C fornisce la definizione delle interfacce DOM mediante l'IDL di CORBA ma rende disponibile anche il binding con Java che è contenuto nel package `org.w3c.dom`.

Riprendendo la todo-list precedente si ha la rappresentazione DOM come mostrato in fig. 4.1.

Figura 4.1 – *Documento DOM corrispondente alla todo-list.*



DOM identifica

- le interfacce e gli oggetti usati per la rappresentazione dei documenti;
- la semantica di tali interfacce e oggetti;
- relazioni e collaborazioni fra oggetti e interfacce.

DOM è stato sviluppato all'interno del W3C e le specifiche sono suddivise in due parti: DOM Level 1 e DOM Level 2. DOM1 definisce le interfacce principali per la manipolazione dei documenti e si compone dei moduli Core e HTML, corrispondenti ai moduli `org.w3c.dom` e `org.w3c.dom.html`. DOM2 estende DOM1 con diversi moduli per funzionalità più avanzate come gestione di diverse view di un documento, stylesheet, eventi e attraversamento della struttura di un documento.

Al momento il W3C sta lavorando sulle specifiche di DOM Level 3 che prevedono, tra l'altro, funzionalità di caricamento e salvataggio di documenti.

Le interfacce di DOM

Si approfondiranno ora le interfacce principali di DOM che, nelle implementazioni Java, sono contenute nel package `org.w3c.dom`.

Come accennato in precedenza, un documento viene rappresentato come un insieme di nodi ed è quindi naturale che l'interfaccia principale sia `Node`. Questa rappresenta un generico nodo nella struttura gerarchica di un documento e definisce i metodi per accedere ad altri nodi nel documento. Particolari tipi di nodi sono rappresentati da interfacce che estendono `Node`; segue ora un elenco delle interfacce principali:

- `Attr`: rappresentazione di un attributo in un elemento. Questo è un caso particolare di nodo in quanto, pur estendendo l'interfaccia `Node`, un attributo non viene considerato dal DOM come nodo figlio dell'elemento che descrive. Si ha quindi che i metodi che accedono ai nodi dell'albero (come `getChildNodes()`) restituiscono `null`;
- `CharacterData`: rappresenta una sequenza di caratteri all'interno del documento ed espone metodi per la gestione di stringhe. Sono definite tre interfacce che la estendono: `Text`, `CDATASection` e `Comment`;
- `Text`: questa interfaccia estende `CharacterData` e rappresenta il testo contenuto all'interno di un elemento o di un attributo. Se il testo non contiene marcatori allora è compreso all'interno di un unico nodo `Text`, altrimenti ne viene eseguito il parsing e lo si inserisce in una lista di nodi;
- `CDATASection`: interfaccia che estende `Text` e rappresenta un nodo che può contenere testo organizzato con marcatori;

- Comment: estende CharacterData e rappresenta il contenuto di un commento inserito fra le sequenze di caratteri <!-- e -->;

- Element: interfaccia per la gestione di un elemento e dei corrispondenti attributi. Si consideri ad esempio il seguente documento XML

```
<ROOTELEMENT>
  <ELEMENT1>
  </ELEMENT1>
  <ELEMENT2>
    <SUBELEMENT1>
    </SUBELEMENT1>
  </ELEMENT2>
</ROOTELEMENT>
```

  Il corrispondente documento DOM avrà un elemento radice ROOTELEMENT, quindi due elementi child ELEMENT1 ed ELEMENT2; quest'ultimo a sua volta ha un elemento child SUBELEMENT1.

Per accedere al tipo di un nodo si usa il metodo getNodeType() dell'interfaccia Node.

Un'altra fondamentale interfaccia del DOM è Document, che come accennato in precedenza rappresenta l'intero documento XML. L'interfaccia Document fornisce inoltre i metodi factory per creare gli altri oggetti che compongono il documento come elementi e nodi di testo. Per la documentazione delle altre interfacce si rimanda a [5].

Gestione di documenti DOM

Utilizzare parser che implementano le specifiche JAXP permette di scrivere codice indipendente dallo specifico parser utilizzato; per DOM si utilizzano le classi

- javax.xml.parsers.DocumentBuilderFactory: classe factory usata per creare un parser;

- javax.xml.parsers.DocumentBuilder: il parser vero e proprio.

Una volta ottenuto un oggetto DocumentBuilder, si può ottenere un documento vuoto con newDocument() oppure eseguire il parsing del documento con varie versioni del metodo parse() per leggere i dati da un file, da un InputSource, da un InputStream o specificando l'URI della risorsa. Il seguente esempio mostra come creare il corrispondente documento DOM della todo-list via codice usando JAXP:

```
import org.w3c.dom.*;
import javax.xml.parsers.*;

...
```

```
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    DocumentBuilder parser = factory.newDocumentBuilder();
    Document doc = parser.newDocument();

    // Creazione dell'elemento root del documento, identificato dal tag <todolist>
    Element root = doc.createElement("todolist");
    doc.appendChild(root);

    // Creazione ed inserimento del primo nodo
    Element item = doc.createElement("item");
    Element number = doc.createElement("number");
    number.appendChild(doc.createTextNode("1"));
    item.appendChild(number);
    Element priority = doc.createElement("priority");
    priority.appendChild(doc.createTextNode("6"));
    item.appendChild(priority);
    Element description = doc.createElement("description");
    description.appendChild(doc.createTextNode("Leggere la posta"));
    item.appendChild(description);
    Element state = doc.createElement("state");
    state.appendChild(doc.createTextNode("2"));
    item.appendChild(state);

    root.appendChild(item);

    // il codice per gli altri elementi è analogo
    // ...
  …
```

Purtroppo JAXP non include, al momento, un meccanismo di serializzazione di un documento DOM; in questo caso è quindi necessario ricorrere ad API proprietarie aspettando le prime implementazioni di DOM3. Nel seguito, nella sezione di approfondimento e banchmark, si vedrà un esempio di serializzazione via JAXP usando una trasformazione XSLT.

Attraversamento

Una volta creato un documento DOM, via codice o mediante parsing, serve un modo per accedere ai dati in esso contenuti partendo dalla radice dell'albero. I figli di un nodo sono contenuti in oggetti che implementano `org.w3c.dom.NodeList`; la lunghezza della lista è esposta dal metodo `getLength()` mentre per accedere ai figli si usa il metodo `item()` specificando un indice (da `0` a `getLength()`-1), analogamente agli array. Il metodo `item()` restituisce un'istanza di `Node`; quindi per ottenere il nome dell'elemento si usa il metodo `getNodeName()`, mentre per accedere ad esso o impostarne il valore si hanno i metodi `getNodeValue()` e `setNodeValue()`. Si osservi che, nel caso in cui il nodo non sia terminale ma contenga a sua volta altri nodi, il metodo `getNodeValue()` restituirà `null`.

Il seguente codice mostra come attraversare i nodi di un documento mediante un algoritmo di visita DFS (depth-first search):

```
void visitDFS(Node node, int level, Writer writer) {
  try {
    // indentazione
    for (int i=0; i<=level; i++) {
      writer.write(" ");
  }

  // stampa del contenuto del nodo
  writer.write(node.getNodeName());
  writer.write("\n");
  writer.flush();

  // per ogni nodo figlio si richiama ricorsivamente
  // questa funzione
  NodeList list = node.getChildNodes();
  for(int i=0; i<list.getLength(); i++) {
    visitDFS(list.item(i), level+1, writer);
  }
  } catch (IOException e) {
    System.err.println("IO exception: " + e.getMessage());
  }
}
```

JDOM

Le sezioni precedenti hanno presentato SAX e DOM, due API standard: esistono quindi diverse implementazioni in vari linguaggi (Perl, Python, linguaggi COM e .NET, …). In questa sezione si presenterà JDOM [6], una API sviluppata appositamente per gestire documenti XML in Java.

JDOM è una API open source, sviluppata da Brett McLaughlin e Jason Hunter, giunta al momento alla versione beta 8. L'idea alla base di JDOM è quella di fornire uno strumento per la gestione di documenti XML con i meccanismi che si aspetterebbe uno sviluppatore Java. Si osservi che, nonostante JDOM sia rivolto agli sviluppatori Java, è comunque una API che interagisce con SAX e DOM; i programmi sviluppati con JDOM possono quindi ricevere in input documenti XML da un parser SAX/DOM e analogamente possono generare un documento DOM o uno stream di eventi SAX.

Un documento JDOM

Un documento XML viene rappresentato come un'istanza della classe `org.jdom.Document`; JDOM fornisce anche l'analogo delle interfacce DOM come classi Java ma si osservi che la gerarchia di

classi JDOM non ha alcuna relazione diretta con DOM; ad esempio la classe Node di JDOM non implementa l'interfaccia Node di DOM.

La prima caratteristica Java-oriented di JDOM che lo sviluppatore incontra è il poter creare gli oggetti direttamente col proprio costruttore, senza la necessità di utilizzare metodi factory. Per creare ad esempio un oggetto org.jdom.Element che rappresenta un tag <Document> è quindi sufficiente scrivere

```
Element e = new Element("Document");
```

Parsing

Per il parsing di documenti XML, JDOM fornisce i *builder*, oggetti che costruiscono un documento a partire da diverse sorgenti dati come file, InputStream e InputSource. Sono definiti due tipi di builder: DOMBuilder e SAXBuilder. Come è evidente dal nome, DOMBuilder carica un documento a partire da un oggetto org.w3c.dom.Document mentre SAXBuilder sfrutta un parser SAX ed è quindi più performante. Il seguente codice mostra come usare un SAXBuilder per eseguire il parsing di un file istanziando un documento JDOM:

```
SAXBuilder builder = new SAXBuilder(true);
Document doc = builder.build(new File(args[0]));
// ...
```

Le classi SAXBuilder e DOMBuilder hanno vari costruttori in overloading che permettono di specificare quale parser usare (il default è Xerces) e se eseguire o meno la validazione del documento (il default è false).

I/O di documenti JDOM

Avere un potente framework per l'elaborazione di documenti XML è importante; ma è altrettanto importante gestire l'I/O di documenti in maniera pratica.

Per quanto riguarda l'input di documenti le varie versioni in overloading del metodo build() coprono un ampio spettro di possibilità. La classe DOMBuilder permette inoltre di creare un documento JDOM a partire da un documento DOM.

L'aspetto più interessante riguarda la gestione dell'output che, come visto, in DOM è abbastanza complicato. La classe org.jdom.output.XMLOutputter si occupa appunto di questo e presenta un pratico metodo output() che accetta come argomenti un documento JDOM e un java.io.OutputStream. Ad esempio, per serializzare un documento su standard output, bastano le seguenti due righe di codice

```
XMLOutputter outputter = new XMLOutputter();
outputter.output(doc, System.out);
```

Per ottenere una stringa si usa invece il metodo outputString():

```
XMLOutputter outputter = new XMLOutputter();
String str = outputter.outputString(doc);
```

La classe SAXOutputter permette di generare uno stream di eventi SAX:

```
ContentHandler handler = ...;
SAXOutputter outputter = new SAXOutputter(handler);
outputter.output(doc);
```

Per ottenere un org.w3c.dom.Document a partire da un documento JDOM si usa invece l'apposita classe DOMOutputter:

```
org.jdom.Document jdoc = ...;
org.w3c.dom.Document doc;

DOMOutputter outputter = new DOMOutputter();
doc = outputter.output(jdoc);
```

Costruzione di un documento JDOM

Facendo riferimento al documento XML di todolist, ecco un esempio di costruzione di un documento JDOM in maniera "manuale", ovvero senza eseguire il parsing di una qualche sorgente dati, e di sua serializzazione su standard output:

```
import java.io.*;
import org.jdom.*;
import org.jdom.output.*;

public class JDOMCreation {
  public static void main(String[] args) {
    Element root = new Element("todolist"), item;
    Document doc = new Document(root);
    item = buildItem("1", "6", "Leggere la posta", "2");
    root.addContent(item);
    item = buildItem("2", "9", "Riunione", "2");
    root.addContent(item);
    item = buildItem("3", "8", "Andare a correre nel parco", "1");
    root.addContent(item);
    try {
      // serializzazione su standard output
      XMLOutputter outputter = new XMLOutputter();
      outputter.setNewlines(true);
      outputter.setIndent("   ");
      outputter.output(doc, System.out);
    } catch(IOException e) {
```

```
            System.err.println("Errore durante la serializzazione del documento");
            e.printStackTrace();
        }
    }

    private final static Element buildItem(String number, String priority, String description, String state) {
        Element item = new Element("item"),
        numberEl = new Element("number"),
        priorityEl = new Element("priority"),
        descriptionEl = new Element("description"),
        stateEl = new Element("state");

        numberEl.setText(number);
        priorityEl.setText(priority);
        descriptionEl.setText(description);
        stateEl.setText(state);

        item.addContent(numberEl);
        item.addContent(priorityEl);
        item.addContent(descriptionEl);
        item.addContent(stateEl);

        return item;
    }
}
```

Il metodo buildItem() costruisce un elemento JDOM corrispondente a un item:

- si crea un elemento item;
- si creano i vari elementi corrispondenti ai dati di un'attività (numero, priorità, descrizione e stato);
- si impostano i valori mediante il metodo setText() di Element;
- si "appendono" gli elementi contenenti i dati all'elemento item, ritornato poi come risultato.

Si noti come, a differenza di DOM, il valore di un elemento non venga gestito come un nodo figlio da appendere, ma in maniera molto più pratica mediante un apposito metodo, come fosse una sua proprietà.

Accesso agli elementi di un documento

Il modello a oggetti di JDOM permette di navigare la struttura di un documento in maniera molto semplice; ad esempio, dato un Document, è possibile accedere all'elemento radice con:

```
Element root = doc.getRootElement();
```

A questo punto si può accedere alla lista di tutti i nodi figli

```
List childrenList = root.getChildren();
```

o a tutti i nodi figli aventi lo stesso nome

```
List childrenList = root.getChildren("CHAPTER");
```

Le precedenti linee di codice sono molto semplici e allo stesso tempo dimostrano quanto JDOM sia un tool vicino allo sviluppatore Java: il metodo getChildren() restituisce un oggetto java.util.List che lo sviluppatore può utilizzare con gli idiomi a cui è abituato. Le collezioni restituite dai metodi di JDOM sono inoltre aggiornabili e le modifiche si ripercuotono sul Document corrispondente. Ad esempio con

```
childrenList.add(new Element("PARAGRAPH"));
```

si aggiunge un nodo discendente all'elemento CHAPTER.

Per accedere ad un singolo nodo figlio si usa il metodo getChild(); in questo caso si accede al primo nodo CHAPTER partendo dalla radice del documento

```
Element e = root.getChild("CHAPTER");
```

Per accedere al contenuto di un elemento si usa il metodo getContent()

```
String content = e.getContent();
```

Utilizzando DOM si sarebbe invece dovuto scrivere

```
String content = e.getFirstChild().getNodeValue();
```

Il seguente esempio mostra, una volta letto il documento XML visto in precedenza e creato il corrispondente documento JDOM, come accedere agli elementi in esso contenuti:

```
import org.jdom.*;
import org.jdom.input.SAXBuilder;
import java.io.File;
import java.util.*;

public class JDOMTest {
  public static void main(String[] args) {
    if (args.length != 1) {
      System.out.println("Utilizzo: java com.mokabyte.javaxml.jdom.JDOMTest <file>");
      return;
    }
```

```
    try {
      SAXBuilder builder = new SAXBuilder();
      Document doc = builder.build(new File(args[0]));
      List children = doc.getRootElement().getChildren();
      String number, priority, description, state;
      Iterator iterator = children.iterator();
      while(iterator.hasNext()) {
        Element item = (Element)iterator.next();

        number = item.getChildText("number");
        priority = item.getChildText("priority");
        description = item.getChildText("description");
        state = item.getChildText("state");

        System.out.println("Attivita' " + number);
        System.out.println("\tPriorita': " + priority);
        System.out.println("\tDescrizione: " + description);
        System.out.println("\tStato: " + state);
      }
    } catch(JDOMException e) {
      System.err.println("Errore durante il parsing del documento " + args[0]);
      e.printStackTrace();
    }
  }
}
```

L’output del programma precedente è

```
Attivita' 1
    Priorita': 6
    Descrizione: Leggere la posta
    Stato: 2
Attivita' 2
    Priorita': 9
    Descrizione: Riunione
    Stato: 2
Attivita' 3
    Priorita': 8
    Descrizione: Andare a correre nel parco
    Stato: 1
```

Attributi

Nel caso in cui l’elemento abbia un attributo lo si può ottenere come stringa

```
String attrValue = e.getAttributeValue("name");
```

o come istanza della classe Attribute

```
Attribute attribute = e.getAttribute("name");
String value = attribute.getValue();
```

È inoltre possibile accedere al valore tipizzato di un attributo mediante una serie di metodi specifici. Ad esempio, per ottenere il valore di un attributo numerico è possibile scrivere

```
int length = attribute.getAttribute("length").getIntValue();
```

Per impostare il valore di un attributo si scriverà invece

```
e.setAttribute("height", "20");
```

Conclusioni

JDOM è uno strumento veramente interessante per lo sviluppatore Java:

- il suo modello a oggetti è molto più pratico di DOM. Ad esempio il poter creare gli oggetti direttamente con il costruttore è sicuramente più immediato che usare metodi factory;
- l'utilizzo delle collection semplifica inoltre notevolmente il codice necessario ed è un idioma di programmazione molto vicino al programmatore Java.

Il futuro di JDOM è molto promettente anche perché il progetto è stato proposto come Java Specification Request e potrebbe essere incluso in una delle prossime versioni di Java.

Data binding

Introduzione

Molte volte l'utilizzo di API come SAX, DOM e JDOM non è adatto perché si ha la necessità di un modello completo a oggetti per il dominio applicativo in esame; si vuole cioè

- mappare un documento XML in una gerarchia di oggetti Java;
- eseguire una qualche logica applicativa sugli oggetti;
- eventualmente trasformare la gerarchia di oggetti ancora in un documento XML da salvare su DB oppure da inviare a un'altra applicazione.

Il data binding risponde proprio a queste esigenze e i vari tool disponibili mettono a disposizione gli strumenti per risolvere i problemi appena presentati.

Questa sezione mostrerà innanzitutto i princìpi alla base del data binding, come i requisiti e alcuni idee alla base di una possibile implementazione. Si presenterà quindi un esempio reale di utilizzo di questa tecnologia; tale esempio sarà alla base della rassegna di tool/framework disponibili nel mondo Java: Castor del Gruppo ExoLab, Zeus del progetto Enhydra e JAXB (Java Architecture for XML Binding) di Sun.

Data binding XML

È possibile trovare un'analogia fra concetti in XML e in un linguaggio di programmazione a oggetti come Java (o anche altri):

- descrizione di un tipo: rappresentata con schema o DTD in XML e classi o interfacce in Java;
- istanza di un tipo: documenti XML e oggetti Java.

Queste analogie sono alla base dei requisiti di uno strumento di data-binding:

- disponibilità di un compilatore di schema/DTD, in grado di generare classi Java a partire da uno schema o da un DTD;
- funzionalità di marshalling, per trasformare in XML un oggetto Java, e di unmarshalling per istanziare un oggetto Java con i dati di un documento XML.

Tali considerazioni devono poi essere valide per una gerarchia di oggetti Java.

Considerando un compilatore di schema/DTD, il primo problema da affrontare è la compatibilità fra i dati contenuti nel documento XML e l'interfaccia della corrispondente classe Java (si tralasceranno i problemi tecnici relativi alla generazione di codice).

Si supponga di avere nel documento un tag per rappresentare un codice con contenuto alfanumerico, ad esempio <Code>C024</Code>; la classe Java dovrà quindi avere una coppia di metodi get/set per l'attributo codice, ovvero

```
public void setCode(String code);
public String getCode();
```

Purtroppo le cose non sono sempre così facili; si consideri ora il caso di una stringa che può avere solo un numero limitato, ovvero una enumerazione, di valori possibili come ad esempio gli stati di un'attività

```
<xsd:simpleType name="stateType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="started"/>
    <xsd:enumeration value="inProgress"/>
```

```
        <xsd:enumeration value="suspended"/>
        <xsd:enumeration value="finished"/>
        </xsd:restriction>
    </xsd:simpleType>
```

Il corrispondente idioma di programmazione Java è una classe che enumera i valori possibili come costanti pubbliche

```
public class State {
  public final String STARTED="started";
  public final String IN_PROGRESS="inProgress";
  public final String SUSPENDED="suspended";
  public final String FINISHED="finished";
}
```

La classe Java corrispondente allo schema avrà una coppia di metodi setState() / getState() è sarà possibile impostare lo stato in modo type safe con

```
activity.setState(State.STARTED);
```

In questo modo però non si ha alcun modo di verificare che il metodo setState() venga richiamato con valori legali. Ci sono diverse alternative:

- generare il metodo setState() in modo che includa il controllo sui valori possibili

```
public void setState(String state) {
  if (state != State.STARTED && state != State.IN_PROGRESS &&
                                    state != State.SUSPENDED && state != State.FINISHED) {
    throw IllegalArgumentException("Invalid state.");
  } else {
    this.state = state;
  }
}
```

- definire un'ulteriore classe State per rappresentare gli stati;
- includere un ulteriore metodo validate() che esegue un controllo su tutti gli attributi della classe in base ai vincoli definiti nello schema.

Per quanto riguarda gli aspetti di marshalling/unmarshalling, lo strumento chiave è rappresentato dalla Reflection API e una trattazione completa degli aspetti tecnici richiederebbe troppo spazio. È possibile trovare in [12], [13], [14], [15] una serie di articoli che spiegano in dettaglio la realizzazione di un tool di data binding.

Esempio

Le sezioni successive presenteranno una rassegna di tool di data-binding XML. Per poterli confrontare si presenterà ora l'esempio di riferimento basato sul documento di todo-list. Per la generazione delle classi Java sono necessari lo schema XML

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="todolist">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="number" type="xsd:integer" minOccurs="1" maxOccurs="1"/>
              <xsd:element name="priority" type="xsd:integer" minOccurs="1" maxOccurs="1"/>
              <xsd:element name="description" type="xsd:string" minOccurs="1" maxOccurs="1"/>
              <xsd:element name="state" type="xsd:integer" minOccurs="1" maxOccurs="1"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

oppure il DTD del documento

```
<!ELEMENT todolist ( item* ) >
<!ELEMENT item (number, priority, description, state) >
<!ELEMENT number (#PCDATA) >
<!ELEMENT priority (#PCDATA) >
<!ELEMENT description (#PCDATA) >
<!ELEMENT state (#PCDATA) >
```

Con ognuno dei tool/framework presentati si procederà nel seguente modo:

- si genereranno le classi corrispondenti alle entità gestite nel documento di todo-list;
- si popolerà una collezione di oggetti a partire dal documento XML precedente;
- si modificheranno gli oggetti nella collezione;
- verrà creato infine un nuovo file XML a partire dalla collezione modificata.

Castor XML

Castor è un framework sviluppato dal gruppo Exolab che implementa

- data binding XML;
- una propria versione JDO, *Java Database Object*, diversa dalla specifica di Sun per la gestione della persistenza degli oggetti Java;
- gestione di directory LDAP.

In questa sezione si valuteranno solo le caratteristiche del framework di data binding XML ma si tenga comunque presente che l'interoperabilità fra JDO, documenti XML e LDAP fornisce a un framework di questo tipo un notevole valore aggiunto.

Il generatore di codice di Castor permette di generare classi Java a partire da uno schema che rispetti la Candidate Recommendation (10/24/2000). Oltre alle classi Java vengono generati anche i class descriptor.

Dopo aver incluso nel classpath castor-0.9.jar e xerces.jar si invoca il generatore di codice nel modo seguente

```
java org.exolab.castor.builder.SourceGenerator -i todolist.xsd
```

Verranno generati i file Todolist.java, TodolistDescriptor.java, Item.java e ItemDescriptor.java.

Todolist.java e Item.java sono le classi Java che rappresentano le entità definite nello schema, mentre TodolistDescriptor.java e ItemDescriptor.java sono, nella terminologia di Castor, i corrispondenti class descriptor. Un class descriptor contiene le informazioni sulla struttura della classe che il framework utilizza durante le fasi di marshalling e unmarshalling. Ci sono quattro modi di generare un class descriptor:

- utilizzando il generatore di codice, come in questo caso;
- implementando l'interfaccia org.exolab.castor.xml.XMLClassDescriptor;
- fornendo al framework un file di configurazione XML;
- lasciando che il framework "deduca" le informazioni sulla classe mediante introspezione, ovvero reflection, sugli oggetti Java.

Le prime due modalità vengono attivate a compile-time e offrono vantaggi di prestazioni rispetto alle altre due possibilità: sono basate sul supporto a run-time del framework. Le classi generate, oltre alle property get/set applicative, presentano caratteristiche molto interessanti:

- incapsulano il codice di marshalling/unmarshalling, ovvero ogni classe ha un metodo unmarshal() per popolare un oggetto da un java.io.Reader e due metodi marshal() per trasmet-

tere in output il corrispondente documento XML a un java.io.Writer, a un handler SAX oppure creare un documento DOM.

- le classi contengono inoltre la gestione delle relazioni fra le entità; in questo caso la classe Todolist ha una serie di metodi per accedere alla collezione di oggetti Item ovvero il generatore di codice ha mantenuto la relazione 1:N fra Todolist e Item;

- ogni classe dispone inoltre di un metodo validate() che permette di validare gli oggetti in base allo schema.

Come si vedrà in seguito, tali caratteristiche sono più o meno presenti anche negli altri tool esaminati. Nella compilazione delle classi precedenti si otterranno dei warning dovuti all'utilizzo dell'interfaccia org.xml.sax.DocumentHandler che è stata deprecata in SAX 2 e sostituita dall'interfaccia org.xml.sax.ContentHandler. Segue ora il codice che utilizza le classi generate

```
import java.io.*;

public class TestCastor {
  public static void main(String args[]) throws Exception {
    Reader reader = new FileReader(args[0]);
    // Popola la todolist con i dati specificati nel file XML
    Todolist list = Todolist.unmarshal(reader);
    // Modifica la collezione
    for(int i=0; i<list.getItemCount(); i++) {
      // impostiamo tutte le attività come da iniziare
      list.getItem(i).setState(0);
    }

    // Aggiungiamo una nuova attività
    Item item = new Item();
    item.setNumber(4);
    item.setPriority(10);
    item.setDescription("Leggere MokaByte");
    item.setState(0);
    list.addItem(item);

    // Validazione dei dati
    list.validate();

    // Salva i dati sul file XML
    Writer writer = new FileWriter(args[0]);
    list.marshal(writer);
  }
}
```

La semplicità del codice precedente mostra decisamente tutti i vantaggi del data-binding: eseguire le stesse operazioni usando SAX, DOM o anche JDOM avrebbe richiesto molte più righe di codice delle 23 viste nella classe TestCastor.

L'oggetto Todolist viene popolato dal file XML specificato dalla linea di comando, quindi si modificano tutte le attività della lista impostando il loro stato a 0, come se le attività fossero tutte ancora da iniziare. Si aggiunge poi una nuova attività semplicemente creando un nuovo oggetto e impostando le sue proprietà. Si valida la Todolist ottenuta e infine si aggiorna il documento salvando l'oggetto modificato.

JAXB

Sun ha sviluppato una serie di framework per l'elaborazione di documenti XML in Java (parsing, trasformazioni XSLT, messaging asincrono di documenti XML, XML RPC, …) e uno di questi framework è proprio JAXB, Java Architecture for XML Binding.

Per la generazione di codice si userà il DTD mostrato in precedenza in quanto JAXB al momento in cui questa sezione viene scritta non supporta ancora gli schema, feature comunque prevista nelle prossime versioni. Prima di questo passo è necessario scrivere un ulteriore documento, il JAXB binding schema, sfruttato dal compilatore nella generazione delle classi. L'idea è quella di fornire una maggiore espressività nella definizione dei tipi e separare i dettagli di programmazione dallo schema. Il binding schema permette ad esempio di definire fra le altre cose costruttori personalizzati e i nomi per package, classi e metodi.

Ecco il binding schema per la todo list

```
<?xml version="1.0"?>
<xml-java-binding-schema version="1.0ea">
  <element name="todolist" type="class" root="true" />
  <element name="number" type="value" convert="int"/>
  <element name="priority" type="value" convert="int"/>
  <element name="state" type="value" convert="int"/>
</xml-java-binding-schema>
```

Il binding schema specifica innanzitutto che l'elemento radice del documento è todolist, quindi vengono specificate le informazioni sul fatto che gli elementi number, priority e state sono interi. Senza queste informazioni il generatore di codice avrebbe gestito tali valori come stringhe. A questo punto, dati il DTD e il binding schema, si è in grado di generare il codice mediante lo schema compiler, incluso nella libreria jaxb-xjc-1.0-ea.jar; per la todolist vengono generate due classi: Todolist.java e Item.java.

Come nel caso di Castor anche le classi generate da JAXB incapsulano il codice di gestione delle relazioni e di validazione. Per quanto riguarda l'unmarshalling, vengono generati metodi che permettono di popolare l'oggetto a partire da un InputStream o da un documento DOM. Per leggere i dati non viene utilizzato un parser completo, in quanto alcuni controlli di validità vengono lasciati al codice generato: a basso livello si utilizza l'oggetto javax.xml.marshal.XMLScanner che permette di scorrere il contenuto del documento con un meccanismo di pattern matching.

Il marshalling del documento viene eseguito su un OutputStream; con la versione attuale non è possibile generare stream SAX o documenti DOM.

Ecco il codice che utilizza le classi generate:

```
import java.io.*;
import java.util.*;
import javax.xml.bind.*;
import javax.xml.marshal.*;

public class TestJAXB {
  public static void main(String args[]) throws Exception {
    File inputFile = new File(args[0]);
    FileInputStream fis = new FileInputStream(inputFile);

    // Popola la todolist con i dati specificati nel file XML
    Todolist list = Todolist.unmarshal(fis);

    List todolist = list.getItem();

    // Modifica la collezione
    for(Iterator i=todolist.iterator(); i.hasNext(); ) {
      Item item = (Item)i.next();
      // impostiamo tutte le attività come da iniziare
      item.setState(0);
    }

    // Aggiungiamo una nuova attività
    Item item = new Item();
    item.setNumber(4);
    item.setPriority(10);
    item.setDescription("Leggere MokaByte");
    item.setState(0);

    todolist.add(item);

    // Validazione dei dati
    list.validate();

    // Salva i dati sul file XML
    File outFile = new File(args[1]);
    FileOutputStream fos = new FileOutputStream(outFile);
    list.marshal(fos);
  }
}
```

Il codice è del tutto analogo a quello usato con Castor e non verrà ulteriormente commentato.

Enhydra Zeus

Zeus è un tool sviluppato all'interno del progetto Enhydra, un application server Java open source. È interessante notare che fra i suoi sviluppatori figura Brett McLaughlin, molto attivo come scrittore tecnico di articoli e libri sulla programmazione XML in Java.

La versione di Zeus utilizzata, 1.0 beta 3.1, permette di generare il codice a partire da un DTD e anche da uno schema. Lo schema visto in precedenza ha però dato luogo ad alcuni problemi nel codice generato in quanto, tra le altre cose, Zeus non ha riconosciuto la struttura gerarchica ed è stato quindi necessario modificare lo schema definendo elementi separati. Per questi motivi il test è stato fatto utilizzando il DTD.

Per lanciare il generatore di codice è necessario includere le librerie zeus.jar, dtdparser113.jar, jdom.jar e xerces.jar; quindi si esegue

```
java org.enhydra.zeus.util.DTDSourceGenerator -constraints=todolist.dtd
```

Una peculiarità del codice generato è che non necessita di alcun supporto a run-time per il marshalling/unmarshalling; non deve quindi sorprendere che per questo semplice esempio si abbiano ben 14 classi generate. Per ogni elemento definito vengono infatti generate un'interfaccia (es. Todolist.java e Item.java), con i metodi di gestione delle relazioni e di marshalling, e un handler SAX (es. TodolistImpl.java e ItemImpl.java) che implementa i metodi dell'interfaccia oltre che ovviamente quelli di unmarshalling via SAX. Esaminando le classi, si può vedere come il marshalling venga invece eseguito mediante concatenazione di stringhe. Viene generata una coppia di classi anche per ogni altro elemento del documento, quindi anche per number, priority, description e state dell'attività. Le classi generate non contengono, a differenza di Castor e JAXB, metodi di validazione.

Oltre alle precedenti, vengono generate due ulteriori classi: Unmarshallable.java, interfaccia che estende org.xml.sax.ContentHandler utilizzata per rappresentare i nodi durante la fase di unmarshalling, e TodolistUnmarshaller.java che contiene i metodi statici per l'unmarshalling dei dati. Ecco il codice di esempio che utilizza le classi generate da Zeus:

```
import java.io.*;
import java.util.*;

public class TestZeus {
  public static void main(String args[]) throws Exception {
    File inputFile = new File(args[0]);

    // Popola la todolist con i dati specificati nel file XML
    Todolist list = TodolistUnmarshaller.unmarshal(inputFile);

    List todolist = list.getItemList();

    // Modifica la collezione
    for(Iterator i=todolist.iterator(); i.hasNext(); ) {
      Item item = (Item)i.next();
      // impostiamo tutte le attività come da iniziare
      State state = new StateImpl();
      state.setValue("0");
      item.setState(state);
    }
```

```
        // Aggiungiamo una nuova attività
        Item item = new ItemImpl();
        NumberImpl number = new NumberImpl();
        number.setValue("4");
        item.setNumber(number);
        Priority priority = new PriorityImpl();
        priority.setValue("10");
        item.setPriority(priority);
        Description description = new DescriptionImpl();
        description.setValue("Leggere MokaByte");
        item.setDescription(description);
        State state = new StateImpl();
        state.setValue("0");
        item.setState(state);

        todolist.add(item);

        // Salva i dati sul file XML
        list.marshal(new File(args[1]));
    }
}
```

La differenza principale rispetto agli esempi precedenti è data dal fatto che anche gli attributi della classe Item vengono rappresentati da classi, ognuna delle quali ha una coppia di metodi getValue()/setValue() per accedere e impostare il valore. Poiché il codice è stato generato da un DTD e quindi senza informazioni sui tipi, tutti i valori vengono rappresentati come stringhe.

Conclusioni

Questa sezione ha introdotto i concetti di data binding XML, discusso alcuni particolari tecnici e presentato una rassegna di alcuni tool disponibili: Castor, Zeus e JAXB. Fra questi Castor sembra sicuramente il più maturo, mentre la principale limitazione di JAXB è il mancato supporto agli schema. Da questo punto di vista la necessità di scrivere in JAXB due documenti, il DTD e il binding schema, è da una parte un problema ma dall'altra offre maggiori possibilità di personalizzazione nel codice generato. Infatti anche Castor permette di parametrizzare la generazione di codice mediante un file di configurazione.

Zeus è un tool ancora in beta ma presenta comunque un'architettura interessante; ad esempio il generatore di codice è pensato per supportare in futuro nuovi linguaggi per la definizione di documenti XML.

Una caratteristica di Castor non presente negli altri tool consiste nell'integrazione con i JDO: in questo modo si ottiene una gestione completa dei passaggi fra tabelle su database, oggetti Java e documenti XML.

Dagli esempi presentati emergono i vantaggi comuni a un qualunque tool di data binding:

- semplicità nell'accesso ai dati;

- non è necessario scrivere codice specifico per le operazioni di input/output sul documento XML;

- il generatore di codice produce inoltre il codice di validazione in base ai constraint definiti nello schema.

La tecnologia di data binding permette quindi di semplificare notevolmente la vita del programmatore, evitandogli di dover scrivere codice monotono di marshalling/unmarshalling per potersi concentrare sulla logica applicativa.

Approfondimenti e benchmark

Applicazioni XML

Realizzare applicazioni basate su XML in Java significa prima di tutto scegliere quali API utilizzare fra le varie disponibili (SAX, DOM, …). Data la disponibilità di diversi tool, è inoltre necessario conoscere cosa offrono le varie implementazioni dei parser e quali è convienente usare, sia in termini di performance che di occupazione di memoria.

Questa sezione non vuole essere solo una presentazione di benchmark di programmi per elaborare documenti XML, innanzitutto perché se ne possono trovare molte altre in rete [17] e poi perché i benchmark vogliono essere un pretesto per discutere gli idiomi di programmazione XML.

Nelle sezioni precedenti sono state presentate le principali API per elaborare documenti XML in Java: SAX, DOM e JDOM. Questa sezione riprende ed estende quella discussione partendo da due esigenze specifiche: creare e leggere documenti XML. Si presenteranno quindi vari esempi per risolvere questi due problemi.

Per gli esempi presentati verranno utilizzati i seguenti parser:

- Xerces 2.1.0 [18]

- Crimson [19] incluso con il JDK 1.4

- JDOM beta 8 [20]

- XML Pull Parser 3 [21]

Generazione di documenti XML

Si vuole ora invece approfondire la fase di costruzione di un documento XML, problema che si presenta ad esempio quando si vogliono esporre in questo formato i dati provenienti da una tabella su DB o da un programma. Si possono individuare le seguenti possibilità:

- concatenazione di stringhe;

- DOM;
- JDOM.

Se i dati fossero disponibili come oggetti Java allora si potrebbe considerare l'utilizzo del data-binding e di un motore di marshalling, ma in questa sede si vuole concentrare l'attenzione sulle tecniche di programmazione per creare un documento XML. SAX è escluso a priori in quanto si tratta di una API di sola lettura.

Per eseguire i confronti si presenteranno classi Java che costruiscono documenti conformi allo schema della todo-list usando le tecniche citate. Il contenuto dei documenti sarà dato da stringhe costanti.

Segue ora la classe StrBuilderBench che costruisce il documento mediante concatenazione di stringhe:

```
public class StrBuilderBench {
  public final String buildXML(int numOfEntities) {
    StringBuffer doc = new StringBuffer("");

    doc.append("<todolist>");
    for (int i=0; i<numOfEntities; i++) {
      doc.append("<item>");
      doc.append("<number>").append(i).append("</number>");
      doc.append("<priority>").append(i).append("</priority>");
      doc.append("<description>").append("dummy").append("</description>");
      doc.append("<state>").append(i).append("</state>");
      doc.append("</item>");
    }
    doc.append("</todolist>");

    // 'serializzazione' del documento come stringa
    String result = doc.toString();

    return result;
  }
}
```

Il codice è estremamente semplice e compatto. Un lato negativo di questo approccio consiste nella necessità di scrivere correttamente i tag di apertura e chiusura: un solo errore, ad esempio l'omissione di uno "/"di fine tag, comprometterebbe la correttezza dell'intero documento. Il codice precedente potrebbe comunque essere ingegnerizzato per limitare questo problema.

Si vuole ora realizzare lo stesso esempio con DOM, ma si pone un problema relativo alla serializzazione del documento come stringa: non è infatti previsto nella API standard un metodo standard per farlo e parser diversi possono quindi fornire metodi diversi per farlo. Ad esempio Xerces fornisce una classe di servizio, come mostra il prossimo esempio:

```
import javax.xml.parsers.*;
import org.w3c.dom.*;
import java.io.*;
import org.apache.xml.serialize.*;

public class XercesDOMBuilderBench {
  public final String buildXML(int numOfEntities) {
    try {
      DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
      DocumentBuilder parser = factory.newDocumentBuilder();
      Document doc = parser.newDocument();
      Element root = doc.createElement("todolist");
      doc.appendChild(root);

      String index = new String();
      for (int i=0; i<numOfEntities; i++) {
        index = String.valueOf(i);
        Element item = doc.createElement("item");
        Element number = doc.createElement("number");
        number.appendChild(doc.createTextNode(index));
        item.appendChild(number);
        Element priority = doc.createElement("priority");
        priority.appendChild(doc.createTextNode(index));
        item.appendChild(priority);
        Element description = doc.createElement("description");
        description.appendChild(doc.createTextNode("dummy"));
        item.appendChild(description);
        Element state = doc.createElement("state");
        state.appendChild(doc.createTextNode(index));
        item.appendChild(state);

        root.appendChild(item);
      }

      // serializzazione del documento come stringa
      OutputFormat format = new OutputFormat(doc);
      StringWriter writer = new StringWriter();
      XMLSerializer serializer = new XMLSerializer (writer, format);
      serializer.asDOMSerializer();
      serializer.serialize(doc);

      String result = writer.toString();

      return result;
    } catch(ParserConfigurationException e) {
      System.err.println("Errore nella configurazione del parser");
      e.printStackTrace();
```

```
      return null;
    } catch(IOException e) {
      System.err.println("Errore di I/O");
      e.printStackTrace();
      return null;
    }
  }
}
```

La costruzione del documento avviene mediante i seguenti passi:

- si crea via JAXP il documento DOM vuoto;
- questo viene usato come factory per creare l'elemento root che viene poi collegato al documento;
- si creano quindi gli elementi corrispondenti alle entità `item`, ai vari attributi (`number`, `priority`, `description` e `state`) e al loro contenuto;
- i vari elementi vengono poi collegati in una struttura gerarchica usando il metodo `appendChild()`.

Il codice precedente è più verboso della corrispondente versione *stringata*, specialmente per quanto riguarda la gestione del contenuto di un nodo che deve necessariamente essere un elemento.

Lo stesso esempio è stato realizzato usando Crimson, che però non dispone di alcuna classe ausiliaria per serializzare il documento. Si è quindi fatto ricorso a un trucco JAXP: usare una trasformazione XSLT che abbia come output uno `StringWriter` ma che non esegua alcuna trasformazione. Per praticità di visualizzazione è stato omesso il codice di costruzione del documento e di gestione delle eccezioni:

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import java.io.*;

public class CrimsonDOMBuilderBench {
  public final String buildXML(int numOfEntities) {
    try {
      // CODICE UGUALE ALL'ESEMPIO PRECEDENTE

      // serializzazione del documento come stringa via JAXP
```

```
        TransformerFactory tFactory = TransformerFactory.newInstance();
        Transformer transformer = tFactory.newTransformer();
        StringWriter writer = new StringWriter();
        transformer.transform(new DOMSource(doc), new StreamResult(writer));
        String result = writer.toString();

        return result;
      } catch(...) {
        ...
      }
    }
  }
```

Dover ricorrere ad una finta trasformazione XSLT è decisamente contorto e la soluzione fornita da Xerces è senz'altro preferibile. In questo caso la soluzione migliore sarebbe comunque un metodo toXML() direttamente fornito dalla classe Document che ritorni la stringa corrispondente. Al momento in cui vengono scritti questi paragrafi, Xerces presenta una prima implementazione di DOM Level 3 con funzionalità di *load and save* di documenti. Viene quindi messo a disposizione un modo standard di scrivere un documento su un OutputStream.

Segue ora lo stesso esempio realizzato con JDOM:

```
import org.jdom.*;
import org.jdom.output.*;

public class JDOMBuilderBench {
  public final String buildXML(int numOfEntities) {
    Element root = new Element("todolist");
    Document doc = new Document(root);

    String index = new String();
    for (int i=0; i<numOfEntities; i++) {
      index = String.valueOf(i);
      Element item = new Element("item"),
        numberEl = new Element("number"),
        priorityEl = new Element("priority"),
        descriptionEl = new Element("description"),
        stateEl = new Element("state");

      numberEl.setText(index);
      priorityEl.setText(index);
      descriptionEl.setText("dummy");
      stateEl.setText(index);

      item.addContent(numberEl);
      item.addContent(priorityEl);
```

```
            item.addContent(descriptionEl);
            item.addContent(stateEl);

            root.addContent(item);
        }

        // serializzazione del documento come stringa
        XMLOutputter outputter = new XMLOutputter();
        String result = outputter.outputString(doc);

        return result;
    }
}
```

Il numero di linee di codice scritto è più o meno lo stesso del corrispondente esempio DOM anche se questa versione JDOM risulta più compatta a vista d'occhio in quanto:

- per creare gli elementi non si usano metodi factory ma costruttori poiché non si devono prevedere estensioni/personalizzazioni del meccanismo di creazione;

- il contenuto di un nodo viene assegnato usando il pratico metodo setText() invece che creando un ulteriore nodo.

Gli esempi precedenti sono stati eseguiti per trarre alcune indicazioni sulle performance dei diversi approcci. Una classe esterna ha invocato ogni metodo buildXML() per 100 volte e sono stati calcolati i tempi medi nella costruzione di documenti di 100, 500, 1000, 1500 e 2000 entità. I risultati ottenuti sono presentati nella tab. 4.2.

Si osservi che questi risultati, come quelli presentati in una sezione successiva, non vogliono avere alcun valore assoluto e sono stati eseguiti su un comune PC. La batteria di test è limitata ma comunque sufficiente per trarre alcuni risultati.

L'implementazione *stringata* risulta decisamente vincente; un altro fatto interessante è dato dalla differenza di tempi fra Crimson e Xerces a vantaggio di quest'ultimo.
Più avanti si approfondiranno i problemi relativi all'input di documenti XML.

Tabella 4.2 – *Tempi medi in millisecondi per la costruzione di documenti di varie entità.*

	100	**500**	**1000**	**1500**	**2000**
Stringhe	2	10	27	64	73
DOM(Xerces)	41	90	148	276	282
DOM(Crimson)	30	151	324	448	536
JDOM	16	97	173	281	338

Lettura di documenti XML

Riprendendo il documento di esempio, l'elaborazione consiste nell'ottenere informazioni sullo stato di avanzamento di tutte le attività alle quali è stata assegnata una certa priorità. Si considereranno dieci valori possibili di priorità, da 1 (bassa priorità) a 10 (alta priorità) e tre valori possibili per gli stati di avanzamento:

1. pianificata (ma non ancora iniziata)
2. iniziata e in corso di svolgimento
3. terminata

Data una priorità si vuole quindi ottenere un output del seguente tipo

```
Attività a priorità 3: ci sono
- 2 attività pianificate
- 5 attività in corso di svolgimento
- 3 attività terminate
```

Una implementazione reale genererebbe un istogramma con queste informazioni ma, ai fini della discussione, non interessa il formato dell'output quanto l'elaborazione eseguita per ottenerlo. Per ottenere i risultati si utilizzerà la stessa struttura dati, un array bidimensionale, incapsulata dalla classe PriorityReport:

```
public class PriorityReport {
  private static final int MAX_PRIORITY = 10;
  private static final int MAX_STATE = 3;
  private static final int PLANNED = 0;
  private static final int IN_PROGRESS = 1;
  private static final int FINISHED = 2;
  private int[][] result;

  public PriorityReport() {
    result = new int[MAX_PRIORITY][MAX_STATE];
  }

  public void addActivity(int priority, int state) {
    result[priority][state]++;
  }

  public void showReport() {
    for(int i=0; i<MAX_PRIORITY; i++) {
      System.out.println("Attività a priorità " + i + ": ci sono");
      System.out.println("-" + result[i][PLANNED] + " attività pianificate");
      System.out.println("-" + result[i][IN_PROGRESS] + " attività in corso di svolgimento");
```

```
      System.out.println("-" + result[i][FINISHED] + " attività terminate");
    }
  }
}
```

Il metodo showReport() mostra le informazioni relative alle varie attività. Questo metodo dovrebbe essere invocato al termine dell'elaborazione anche se in realtà nei test successivi è stato commentato per praticità ma soprattutto perché si tratta di un metodo che non esegue alcuna elaborazione.

SAX

Ecco un handler SAX che intercetta le informazioni su priorità e stati di avanzamento, memorizzandole in un PriorityReport:

```
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class SAXPriorityHandler extends DefaultHandler {
  private String currentElement;
  private int priority;
  private int state;
  private PriorityReport report;

  public SAXPriorityHandler() {
    report = new PriorityReport ();
  }


public void startElement(String uri, String name, String qName, Attributes attributes) throws SAXException {
    currentElement = qName;
  }

  public void characters(char[] ch, int start, int length) throws SAXException {
    if (currentElement.equals("priority")) {
      priority = Integer.parseInt(new String(ch, start, length));
    } else if (currentElement.equals("state")) {
      state = Integer.parseInt(new String(ch, start, length));
    }
  }

  public void endElement(String namespaceURI, String localName, String qName) throws SAXException {
    if (currentElement.equals("item")) {
      report.addActivity(priority, state);
    }
    currentElement = "";
```

```
    }
  }
```

Il codice precedente è abbastanza semplice purché ci si ponga in una logica a eventi che prevede la suddivisione della logica di elaborazione fra i vari metodi dell'handler.

Parsing pull

Una delle caratteristiche principali del parsing SAX consiste nello scrivere il codice di gestione negli handler in diversi metodi corrispondenti agli eventi generati dal parser. Questo può portare a un codice molto complesso da gestire.

L'idea alla base dei parser pull consiste nell'elaborare un documento XML come uno stream di dati, mantendo quindi il codice di gestione del documento nel client che invoca il parser. Il client invoca infatti sul parser un metodo `next()` che restituisce il successivo evento di parsing che il client può elaborare.

È in corso di sviluppo una API comune, XMLPULL V1 API [22], che i vari parser possono implementare analogamente a JAXP. La API è molto semplice ed è composta da:

- `XmlPullParser`: interfaccia che definisce le funzionalità di un parser XPP;
- `XmlPullParserFactory`: classe factory utilizzata per creare la specifica istanze del parser;
- `XmlPullParserException`: eccezione che segnala un errore durante il parsing.

I metodi principali dell'interfaccia `XmlPullParser` sono `next()` e `nextToken()`. Il parser viene considerato come una macchina a stati: si parte dallo stato iniziale `START_DOCUMENT` e si invoca un metodo, `next()`, per passare agli stati successivi:

- `START_TAG`: è stato letto un tag di apertura ed è possibile accedere a varie informazioni come il nome dell'elemento e i suoi attributi.
- `TEXT`: il contenuto di un elemento.
- `END_TAG`: tag di fine elemento.
- `END_DOCUMENT`: fine del documento.

Per avere un set di eventi più granulare in grado di intercettare processing instruction, commenti e altre informazioni sul documento elaborato si usa il metodo `nextToken()`.

Segue ora la versione XPP dell'esempio:

```
import java.io.*;
import org.xmlpull.v1.*;
```

```
public class PullHandler {
  public static void main (String args[]) {
    if (args.length != 1) {
      System.out.println("Utilizzo: java com.mokabyte.javaxml.benchmark.PullHandler <file>");
      return;
    }

    try {
      // creazione del parser pull
      XmlPullParserFactory factory = XmlPullParserFactory.newInstance();
      XmlPullParser parser = factory.newPullParser();

      // configurazione dell'input
      parser.setInput(new FileReader(args[0]));

      int eventType, priority=0, state=0;
      boolean readingPriority = false, readingState = false;
      String content = null;
      PriorityReport report = new PriorityReport();

      // ciclo principale di elaborazione
      while((eventType = parser.next()) != XmlPullParser.END_DOCUMENT) {
        switch(eventType) {
          case XmlPullParser.START_TAG:
            if (parser.getName().equals("priority")) {
              readingPriority = true;
            } else if (parser.getName().equals("state")) {
              readingState = true;
            }
            break;
          case XmlPullParser.TEXT:
            if (readingPriority || readingState) {
              content = parser.getText().trim();
            }
            break;
          case XmlPullParser.END_TAG:
            if (readingPriority) {
              priority = Integer.parseInt(content);
              readingPriority = false;
            } else if (readingState) {
              state = Integer.parseInt(content);
              readingState = false;
            } else if (parser.getName().equals("item")) {
              report.addActivity(priority, state);
            }
        }
      }
```

```
            // output dei risultati
            report.showReport();
        } catch(XmlPullParserException e) {
            System.out.println("Errore durante la configurazione del parser");
            e.printStackTrace();
        } catch(IOException e) {
            System.out.println("Errore durante la lettura del file " + args[0]);
            e.printStackTrace();
        }
    }
}
```

Al momento, le uniche due implementazioni disponibili sono XPP3 [21], utilizzato in questo esempio, e kXML [23]. È stata proposta una JSR per includere tale API in una futura versione di Java.

DOM

Per eseguire la stessa elaborazione con DOM è necessario eseguire una visita dell'albero che rappresenta il documento XML: con un primo ciclo si esamineranno tutti gli elementi item mentre un secondo ciclo interno esaminerà le informazioni di ogni item per accedere ai valori di priorità e stato. Quest'ultimo ciclo può sembrare eccessivo e risulta interessante valutare le alternative:

- usare il metodo `getElementsByTagName()` su ogni oggetto rappresentante un item, ad esempio

  ```
  Node item = ...;
  priority = Integer.parseInt(item.getElementsByTagName("priority").item(0).getNodeValue);
  ...
  ```

  Questo metodo esamina però ogni nodo raggiungibile a partire dall'elemento su cui viene invocato, quindi non si elimina il costo della visita su tutti i figli di item.

- Usare metodi ad accesso posizionale (`getFirstChild()` e `getNextSibling()`) che però richiedono comunque un loop su tutti gli elementi a meno che non si sappia a priori che gli elementi ricercati sono ad esempio nelle prime due posizioni.

Segue ora il codice per l'elaborazione della todo-list con DOM:

```
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;
```

```
public class DOMBenchmark {

  public void readFile(String file) {
    try {
      DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
      DocumentBuilder parser = factory.newDocumentBuilder();
      Document doc = parser.parse(new File(file));
      NodeList list = doc.getDocumentElement().getChildNodes();
      int priority=0, state=0;
      PriorityReport report = new PriorityReport();

      // Loop su tutti gli elementi item
      for(int i=0; i<list.getLength(); i++) {
        Node node = list.item(i);
        NodeList elements = node.getChildNodes();
        if (elements.getLength() > 0) {

          // loop sugli elementi di item per accedere a priorità e stato
          for(int j=0; j<elements.getLength(); j++) {
            Node n = elements.item(j);
            if (n.getNodeName().equals("priority")) {
              priority = Integer.parseInt(n.getFirstChild().getNodeValue());
            } else if (n.getNodeName().equals("state")) {
              state = Integer.parseInt(n.getFirstChild().getNodeValue());
            }
          }
          report.addActivity(priority, state);
        }
      }

    } catch(...) {
      ...
    }
  }
}
```

JDOM

L’analogo codice JDOM sfrutta il fatto che i discendenti di un nodo sono organizzati in un oggetto java.util.List; risulta quindi molto pratico scorrere gli elementi con un Iterator.

```
import java.io.*;
import org.jdom.*;
import org.jdom.input.*;
import java.util.*;
```

```
public class JDOMBenchmark {

  private String saxDriverClass;

  public JDOMBenchmark(String saxDriverClass) {
    this.saxDriverClass = saxDriverClass;
  }

  public void readFile(String file) {
    try {
      SAXBuilder builder = new SAXBuilder(saxDriverClass);
      Document doc = builder.build(new File(file));
      List children = doc.getRootElement().getChildren();
      int priority, state;

      PriorityReport report = new PriorityReport();

      Iterator iterator = children.iterator();
      while(iterator.hasNext()) {
        Element item = (Element)iterator.next();

        priority = Integer.parseInt(item.getChildText("priority"));
        state = Integer.parseInt(item.getChildText("state"));
        report.addActivity(priority, state);
      }
    } catch(…) {
      …
    }
  }
}
```

Questo esempio è sicuramente il più semplice di quelli presentati nelle sezioni precedenti. Si osservi come, mediante il costruttore, viene specificato quale parser SAX utilizzare; in questo modo lo stesso esempio funziona con un qualsiasi parser SAX2.

Tabella 4.3 – *Risultati dei test di elaborazione (tempi medi in millisecondi).*

	100 elementi	**500 elementi**	**1000 elementi**	**1500 elementi**	**2000 elementi**
SAX(Xerces)	35	55	79	103	128
SAX(Crimson)	13	28	46	64	82
Pull	18	35	59	83	107
DOM(Xerces)	57	118	200	276	363
DOM(Crimson)	17	68	152	234	298
JDOM(Xerces)	53	187	359	521	591
JDOM(Crimson)	30	154	303	460	528

Qualche benchmark

Per eseguire i test di elaborazione sono stati creati 5 documenti XML conformi allo schema della todo-list contenenti rispettivamente 100, 500, 1000, 1500 e 2000 elementi. Ogni esempio ha quindi letto ed elaborato il documento 100 volte ed è infine stato calcolato il tempo medio in millisecondi (tab. 4.3).

Il vincitore risulta essere SAX utilizzando Crimson come parser. È interessante la differenza di prestazioni fra Xerces e Crimson in scrittura e lettura.

Validazione di documenti XML

Introduzione

Nelle sezioni precedenti sono state approfondite varie tecniche di programmazione XML in Java. Si vuole ore approfondire il problema della validazione, ovvero come garantire che un documento XML abbia senso dal punto di vista applicativo. Verranno presentate le due principali tecnologie, DTD e Schema, e il loro utilizzo con JAXP.

DTD

Un DTD (Document Type Definition), è un documento che permette di definire gli elementi utilizzabili in un particolare documento XML e la loro struttura. Segue ora il DTD di definizione per la todolist:

```
<!ELEMENT todolist (item*)>
<!ELEMENT item (number,priority,description,state)>
<!ELEMENT number (#PCDATA)>
<!ELEMENT priority (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT state (#PCDATA)>
```

La prima riga indica che l'elemento todolist contiene vari elementi item mentre il carattere * specifica la cardinalità di questa relazione e indica che una todolist comprende 0 o più item. Gli altri possibili qualificatori sono:

- il carattere + per indicare una o più occorrenze di un elemento;
- il carattere ? per indicare un elemento opzionale;
- nessun qualificatore per specificare una e una sola occorrenza di quel dato elemento.

La seconda riga del DTD definisce la struttura di un item come contenente gli elementi number, priority, description e state:

```
<!ELEMENT item (number,priority,description,state)>
```

Infine si specifica che i vari elementi contengono testo, chiamato *parsed character data* (PCDATA) in questo contesto. Il carattere # che precede PCDATA indica che si tratta di una parola chiave e non del nome di un elemento.

Il DTD viene referenziato dal documento con

```
<?xml version="1.0"?>
<!DOCTYPE todolist SYSTEM "todolist.dtd">
<todolist>
  ...
</todolist>
```

In alternativa potrebbe essere incluso direttamente nel documento:

```
<?xml version="1.0"?>
<!DOCTYPE todolist SYSTEM [
  ...
]>
<todolist>
  ...
</todolist>
```

Da quanto visto emergono alcune limitazioni per i DTD come ad esempio le seguenti:

- non è possibile specificare per gli elementi un tipo diverso da PCDATA;
- non si possono definire enumerazioni;
- non si possono definire range di validità;

Solo questi problemi rendono improponibile specificare validazioni complesse mediante DTD.

Schema

Gli schema XML sono uno standard W3C per la definizione della struttura di un documento XML e dei suoi vincoli di validità. L'aspetto particolarmente interessante degli schema è dato dal fatto che si tratta di un linguaggio XML; ecco ad esempio lo schema per la definizione del documento di todolist:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="todolist">
```

```
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="item" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="item">
    <xsd:complexType>
      <xsd:all>
        <xsd:element name="number" type="xsd:integer"/>
        <xsd:element name="priority" type="xsd:integer"/>
        <xsd:element name="description" type="xsd:string"/>
        <xsd:element name="state" type="xsd:integer"/>
      </xsd:all>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Come si può vedere, lo schema permette di definire più informazioni e vincoli rispetto ai DTD:

- per ogni elemento è possibile definire una cardinalità minima e massima mediante gli attributi minOccurs e maxOccurs;

- l'attributo type permette di specificare il tipo di un elemento in maniera dettagliata.

Per maggiori informazioni relative all'utilizzo degli schema si rimanda a [2].

Come per i DTD, è possibile specificare il riferimento ad uno schema nel documento XML mediante un attributo nell'elemento root del documento; la sintassi varia a seconda che il documento usi o meno i namespace, si userà quindi

- xsi:schemaLocation (dove XSI sta per XML Schema Instance) se il documento usa i namespace, ad esempio

```
<todolist
  xmlns="http://www.mytodolist.com"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="todolist.xsd">
  ...
</todolist>
```

- xsi:noNamespaceSchemaLocation se il documento non utilizza i namespace, ad esempio

```
<todolist
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
    xsi:noNamespaceSchemaLocation="todolist.xsd">
    ...
  </todolist>
```

È interessante vedere come associare più schema a uno stesso documento. Si supponga ad esempio di voler includere nel documento todolist informazioni relative a una specifica attività, come quella di scrivere articoli per MokaByte; queste ulteriori informazioni saranno contenute in un altro schema e referenziate dal documento nel modo seguente:

```
<todolist
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="todolist.xsd"
  xsi:schemaLocation="http://www.mokabyte.it/ article.xsd">
  ...
</todolist>
```

Queste prime sezioni hanno sviluppato il discorso della validazione dei documenti XML mantenendolo però indipendente dalla particolare tecnologia.

Un parser in grado di verificare la conformità di un documento a un DTD o a uno schema viene chiamato *validante*. Di seguito si approfondirà l'utilizzo di parser validanti in Java.

Validazione con JAXP

Per poter utilizzare un parser validante con JAXP è necessario configurare opportunamente il factory; il seguente esempio mostra come ottenere un parser SAX validante usando il metodo setValidating() di SAXParserFactory:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setValidating(true);
SAXParser parser = factory.newSAXParser();
DefaultHandler handler = new DummyHandler();
parser.parse(new FileInputStream(docURI), handler);
```

Il DTD deve essere referenziato dal documento come mostrato nelle sezioni precedenti. L'esempio utilizza come handler SAX una classe che implementa solo i metodi di gestione degli errori che saranno mostrati in seguito.

Si osservi che un DTD risulta utile anche usando un parser SAX non validante. Un documento può contenere spazi bianchi, ad esempio per indentare gli elementi in un file di configurazione scritto a mano. Questi spazi verranno inutilmente segnalati all'handler come comune testo e passati al metodo characters(). Specificando un DTD, un parser Java invocherà il metodo ignorableWhiteSpace(), anche se le specifiche SAX in generale non vincolano i parser a questo comportamento.

Per utilizzare gli schema è necessario specificare altre informazioni: innanzitutto, poiché gli schema sono basati sull'utilizzo dei namespace, il parser deve supportare i namespace e si deve

inoltre specificare che si stanno utilizzando gli schema e non i DTD. Queste configurazioni vengono impostate nelle property del parser, come mostra il seguente esempio:

```
private static final String JAXP_SCHEMA_LANGUAGE
= "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
private static final String JAXP_SCHEMA_SOURCE
= "http://java.sun.com/xml/jaxp/properties/schemaSource";
private static final String W3C_XML_SCHEMA = "http://www.w3.org/2001/XMLSchema";
. . .
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setNamespaceAware(true);
factory.setValidating(true);
SAXParser parser = factory.newSAXParser();
parser.setProperty(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
parser.setProperty(JAXP_SCHEMA_SOURCE, new File(schemaURI));
DefaultHandler handler = new DummyHandler();
parser.parse(new FileInputStream(docURI), handler);
```

Per associare più schema al documento è sufficiente definire un array contenente gli URI degli schema e impostare con questo la property .../schemaSource:

```
String[] shemas = {"todolist.xsd", "article.xsd"};
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setNamespaceAware(true);
factory.setValidating(true);
SAXParser parser = factory.newSAXParser();
parser.setProperty("http://java.sun.com/xml/jaxp/properties/schemaLanguage",
                   "http://www.w3.org/2001/XMLSchema");
parser.setProperty("http://java.sun.com/xml/jaxp/properties/schemaSource", shemas);
. . .
```

Quanto visto in precedenza si applica immediatamente anche a un parser DOM. Le property in precedenza impostate sul parser SAX ora sono attributi del factory e si deve comunque impostare un ErrorHandler per intercettare gli errori:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setValidating(true);
DocumentBuilder parser = factory.newDocumentBuilder();
parser.setErrorHandler(new DummyHandler());
Document doc = parser.parse(new FileInputStream(docURI));
```

Se non si imposta un handler per gli errori, allora il parser utilizza un handler di default e a fronte di un errore si ottiene il seguente output:

```
Warning: validation was turned on but an org.xml.sax.ErrorHandler was not set, which is probably not
what is desired. Parser will use a default ErrorHandler to print the first 10 errors.
Please call the 'setErrorHandler' method to fix this.
...
```

Errori di validazione

Per poter intercettare gli errori di validazione con un parser SAX si deve utilizzare un opportuno handler, org.xml.sax.ErrorHandler, che definisce i seguenti metodi:

```
public void warning(SAXParseException e) throws SAXException;
public void error(SAXParseException e) throws SAXException;
public void fatalError(SAXParseException e) throws SAXException;
```

Utilizzando un parser validante viene rilevato un error() se il documento non è valido mentre si solleva un fatalError() per un errore non recuperabile, come nel caso del parsing di un documento non ben formato.

Tutti i metodi rendono disponibile una SAXParseException che permette di recuperare informazioni relative all'errore verificatosi; più precisamente i metodi

- getColumnNumber() e getLineNumber() permettono di risalire alla posizione del testo che ha causato l'errore;
- getPublicId() e getSystemId() restituiscono gli identificatori dell'entità nella quale si è verificato l'errore.

SAXParseException estende inoltre SAXException ed eredita da questa eccezione la possibilità di contenere altre eccezioni.

La classe DummyHandler inclusa negli esempi implementa per semplicità solo i metodi error() e warning():

```
public void error(SAXParseException e) throws SAXParseException {
  System.out.println("Errore di validazione: linea " + e.getLineNumber());
  System.out.println(e.getMessage());
}

public void warning(SAXParseException e) throws SAXParseException {
  System.out.println("Warning: linea " + e.getLineNumber());
  System.out.println(e.getMessage());
}
```

La classe ValidationTest, anch'essa inclusa negli esempi, permette di eseguire il parsing del documento specificato usando un DTD o uno schema per la validazione. Modificando il documento XML di esempio, sostituendo all'elemento item un imprecisato elemento numbero

```
<?xml version="1.0"?>
<!DOCTYPE todolist SYSTEM "file:/C:\Documenti\Articoli\In Lavorazione\validazione\todolist.dtd">
<todolist>
  <item>
    <numbero>1</numbero>
    <priority>6</priority>
  . . .
```

si ottiene il seguente output

```
C:\...>java ValidationTest sax-dtd todolist_dtd.xml
Errore di validazione: linea 5
Element type "numbero" must be declared.
Errore di validazione: linea 9
The content of element type "item" must match "(number,priority,description,state)".
```

Utilizzando lo schema si ottiene invece

```
...
Errore di validazione: linea 4
cvc-complex-type.2.4.a: Invalid content starting with element 'numbero'.
The content must match 'all(("":number),("":priority),("":description),("":state))'.
```

XSLT

Introduzione

Nel paragrafo introduttivo su XML si è accennato a XSL, il linguaggio usato per trasformare i documenti XML. In questo paragrafo si approfondiranno i concetti visti.

XSL [28] è composto da due parti

- **XSL Transformation** (XSLT): un linguaggio per trasformare documenti XML in altri documenti. Si noti che il formato di destinazione potrebbe essere ancora XML oppure qualcosa di diverso, come ad esempio un formato grafico. In questo caso il foglio di trasformazione viene chiamato stylesheet. Questo è comunque solo un esempio dei possibili utilizzi di XSLT.

- **Formatting Objects** (FO): un linguaggio per descrivere il layout del testo. Un documento XSL:FO contiene quindi delle direttive per indicare al motore di rendering come visualizzare il documento. Per maggiori informazioni si rimanda a [29].

Si osservi che trasformazione di documenti è un termine generico che comprende il mapping di un documento XML in una pagina HTML ma anche la trasformazione di un documento da un

dato schema in uno schema diverso. XSL è quindi alla base delle principali applicazioni XML, ovvero web publishing e integrazione di applicazioni.

XSLT

Un documento XSL viene elaborato da un processore che, dati in ingresso un documento XML e un documento XSL, restituisce un nuovo documento XML.

Si vedranno ora alcuni fondamenti di XSLT. Innanzi tutto una trasformazione XSL viene definita in un file XML come il seguente

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
...
</xsl:stylesheet>
```

L'intestazione del documento indica la versione di XSL e il namespace utilizzati, ovvero si dichiara che si useranno dei tag con prefisso xsl e che la definizione di questi tag corrisponde all'URL indicato. È necessario porre molta attenzione al namespace poiché engine XSL diversi potrebbero implementare namespace diversi e incompatibili.

XSLT è un linguaggio dichiarativo; una trasformazione viene quindi specificata con una regola che attraverso un meccanismo di pattern matching viene applicata all'elemento XML di partenza. Una regola viene espressa mediante un template:

```
<xsl:template match="node">
...
</xsl:template>
```

Quando l'engine XSL incontrerà questa regola, allora ricercherà nel documento XML un elemento di nome node e ad esso applicherà la trasformazione. Per attivare la regola precedente è necessario utilizzare il comando

```
<xsl:apply-templates select="node"/>
```

Si consideri ora il seguente documento XML

```
<article>
  <title>XSL and Java</title>
  <author email="doe@doe.com">John Doe</author>
</article>
```

Per accedere al contenuto di un singolo nodo si usa il comando xsl:value-of; il contenuto del nodo title sarà quindi

```
<xsl:value-of select="title"/>
```

Il valore dell'attributo select permette molta libertà nell'accesso agli elementi. È ad esempio possibile accedere agli attributi di un elemento usando il carattere @; il valore dell'attributo email di author è quindi

```
<xsl:value-of select="author/@email"/>
```

Si possono inoltre specificare elementi annidati (article/author), wildcards (*/title) e altri pattern mediante espressioni XPath [30]. XPath è un linguaggio definito dal W3C per individuare le singole parti di un documento XML ed è al momento giunto alla versione 1.0.

Vi sono molti altri comandi in XSLT. Si possono ad esempio esprimere condizioni con i comandi xsl:choose e xsl:if. Il primo permette una scelta multipla fra diversi condizioni, analogamente allo switch in Java

```
<xsl:choose>
  <xsl:when test="condition1">
    ...
  </xsl:when>
  <xsl:when test="condition2">
    ...
  </xsl:when>
  <xsl:otherwise>
    ...
  </xsl:otherwise>
</xsl:choose>
```

mentre xsl:if permette di indicare una sola condizione e nessuna alternativa

```
<xsl:if test="condition">
  ...
</xsl:if>
```

In XSLT è inoltre possibile esprimere cicli con l'istruzione xsl:for-each:

```
<xsl:for-each select="pattern">
  ...
</xsl:for-each>
```

Trasformazioni XSLT via Servlet

Si vedrà ora un esempio completo di utilizzo di XSL. Sia dato il seguente documento, una lista di articoli

```
<?xml version="1.0" ?>
<articles>
  <article>
    <title>XSL and Java</title>
    <author id="1">
      <first-name>John</first-name>
      <last-name>Doe</last-name>
      <email>doe@doe.com</email>
    </author>
  </article>
  <article>
    <title>Distributed Java programming</title>
    <author id="2">
      <first-name>Tod</first-name>
      <last-name>Mann</last-name>
      <email>tod@foo.com</email>
    </author>
  </article>
  <article>
    <title>Java, XML and enterprise application integration</title>
    <author id="3">
      <first-name>Frank</first-name>
      <last-name>Kipple</last-name>
      <email>kipple@bar.com</email>
    </author>
  </article>
</articles>
```

Il documento precedente potrebbe essere il risultato di una query su database; per trasformarlo in una tabella HTML in modo da poterla visualizzare in un browser, si userà il seguente documento XSL

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="/">
    <HTML>
    <BODY>
      <TITLE>Articles' list</TITLE>
      <TABLE BORDER="1">
        <TH>Title</TH><TH>Author</TH><TH>email</TH>
        <xsl:apply-templates select="//article"/>
      </TABLE>
    </BODY>
    </HTML>
  </xsl:template>
```

```
  <xsl:template match="article">
    <TR>
      <TD><x<xsl:template match="article"></TD>
      <TD><xsl:value-of select="title"/></TD>
      <TD><xsl:value-of select="author/first-name"/>
        <xsl:text> </xsl:text>
        <xsl:value-of select="author/last-name"/>
        </TD>
      <TD><xsl:value-of select="author/email"/></TD>
    </TR>
  </xsl:template>
</xsl:stylesheet>
```

Per visualizzare la pagina HTML generata dal precedente file XSL ci sono due possibilità:

- trasferire i file XML e XSL direttamente al browser, ma si tratta di una funzionalità supportata al momento solo da Internet Explorer;
- eseguire la trasformazione sul server e mandare al browser una comune pagina HTML. In questo modo si garantisce il supporto a più browser e, parametrizzando opportunamente la trasformazione XSL, è possibile ad esempio generare codice WML per i browser WAP.

Si vuole ora realizzare come esempio una semplice servlet che genera il codice HTML corrispondente ai file XML e XSL specificati come parametri. Il metodo doGet() della servlet, chiamata XMLServlet, inoltrerà la richiesta al seguente metodo doXSLProcessing()

```
private void doXSLProcessing(HttpServletRequest request, HttpServletResponse response)
                throws IOException, ServletException {

  response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  try {
    // Si ottengono gli URL corrispondenti ai documenti XML e XSL specificati come parametri.
    String xmlUrl = getDocURL("XML", request);
    String xslUrl = getDocURL("XSL", request);

    TransformerFactory factory = TransformerFactory.newInstance();
    InputStream xslt = getServletContext().getResourceAsStream(xslUrl), xml
                        = getServletContext().getResourceAsStream(xmlUrl);
    Transformer transformer = factory.newTransformer(new StreamSource(xslt));
    transformer.transform(new StreamSource(xml), new StreamResult(out));
  } catch(Exception e) {
    out.write(e.getMessage());
  }
}
```

Il metodo, ottenuti gli URL del documento XML da visualizzare e del file XSL da usare per la formattazione, manda il risultato della trasformazione al browser. Per eseguire la trasformazione vera e propria, si usa JAXP che permette di nascondere il riferimento al particolare engine XSLT utilizzato. Le classi StreamSource e StreamResult incapsulano vari possibili formati di input/output come un file, un URI, un InputStream/OutputStream o un Reader/Writer.

Nell'esempio precedente si esegue ogni volta un parsing del documento XSLT, cosa che può penalizzare le prestazioni dell'applicazione. JAXP dà la possibilità di ottenere una versione compilata dello stylesheet chiamata *translet*, riutilizzabile per trasformazioni diverse anche in thread diversi. Si usa a questo proposito la classe Templates:

```
TransformerFactory factory = TransformerFactory.newInstance();
Templates translet = factory.newTemplates(new StreamSource(xslt));
```

La trasformazione si esegue quindi con

```
Transformer transformer = translet.newTransformer();
transformer.transform(new StreamSource(xml), new StreamResult(out));
```

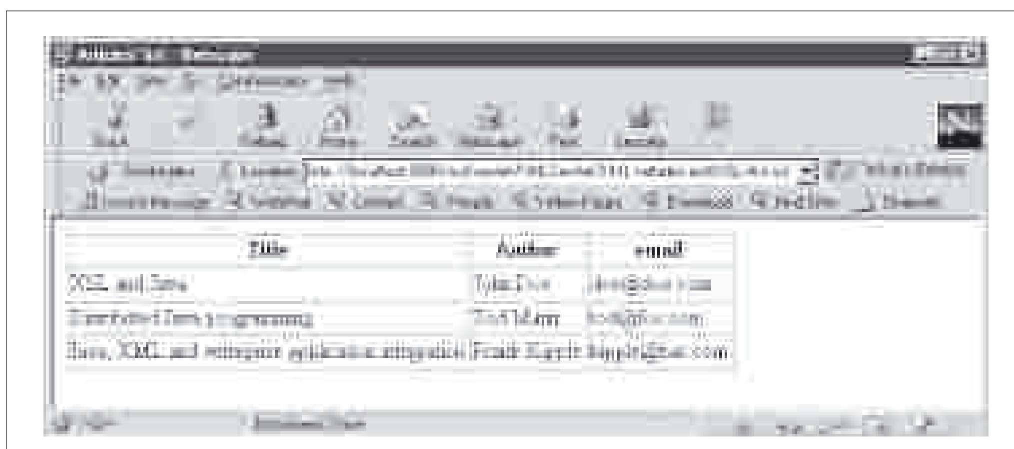
Per attivare la servlet si digita il seguente URL

```
http://localhost/xsl/servlet/XMLServlet?XML=articles.xml&XSL=list.xsl
```

e verrà visualizzata la pagina mostrata in fig. 4.2.

La trasformazione di documenti XML è alla base delle applicazioni che vogliano sfruttare questa tecnologia. Il vantaggio principale di usare XSL consiste nel non dover cablare la logica di trasformazione nel codice. In questo modo è possibile apportare modifiche senza dover ricorrere a ricompilazioni dell'applicazione. Per approfondire l'utilizzo di XSL e XPath con Java si veda [31].

Figura 4.2 – *Risultato della trasformazione XSL applicata dalla servlet.*



Bibliografia

[1] Specifiche W3C di XML, http://www.w3c.org/XML

[2] Schema XML, http://www.w3c.org/Schema

[3] Java XML, http://java.sun.com/xml

[4] Il sito di SAX, http://www.megginson.com/SAX

[5] Il sito del W3C dedicato a DOM, http://www.w3c.org/DOM

[6] Il sito del progetto JDOM, http://www.jdom.org

[7] ELLIOTTE RUSTY HAROLD, *Processing XML with Java*", http://www.cafeconleche.org /books/xmljava/chapters/index.html

[8] WES BIGGS – HARRY EVANS, *Simplify XML Programming with JDOM*, IBM DeveloperWorks

[9] SAM BRODKIN, *Use XML data binding to do your laundry*, in «JavaWorld», gennaio 2002

[10] Castor, http://castor.exolab.org

[11] Zeus Enhydra, http://zeus.enhydra.org

[12] BRETT MCLAUGHLIN, *Objects, objects everywhere*", http://www-106.ibm.com /developerworks/library/data-binding1/?dwzone=xml, luglio 2000

[13] BRETT MCLAUGHLIN, *Make classes from XML data*, http://www-106.ibm.com /developerworks/library/data-binding2/index.html?dwzone=xml, agosto 2000

[14] BRETT MCLAUGHLIN, *From text to byte code*, http://www-106.ibm.com/developerworks /library/data-binding3/index.html?dwzone=xml, settembre 2000

[15] BRETT MCLAUGHLIN, *From bits to brackets*, http://www-106.ibm.com/developerworks /library/data-binding4/index.html?dwzone=xml, ottobre 2000

[16] Sun, Java Architecture for XML Binding, http://java.sun.com/xml/jaxb

[17] THIERRY VIOLLEAU, *Java Technology and XML Part 2: API Benchmarks*, http://developer.java.sun.com/developer/technicalArticles/xml/JavaTechandXML_part2/

[18] Xerces, http://xml.apache.org/xerces

[19] Crimson, http://xml.apache.org/crimson

[20] JDOM, http://www.jdom.org

[21] XML Pull Parser (XPP), http://www.extreme.indiana.edu/xgws/xsoap/xpp/

[22] XMLPULL API, http://www.xmlpull.org

[23] kXML, http://kxml.org/

[24] BRETT MCLAUGHLIN, *Validation with Java and XML schema, Part 1*, http://www.javaworld.com/javaworld/jw-09-2000/jw-0908-validation_p.html

[25] JING XUE – DANIEL COX, *Java and XML: Learn a Better Approach To Data Validation*, http://www.devx.com/java/free/articles/schwartz0402/schwartz0402-1.asp

[26] AHMED ABULSOROUR – SIVA VISVESWARAN, *Business process automation made easy with Java, Part 1*, http://www.javaworld.com/javaworld/jw-09-2002/jw-0906-process_p.html

[27] AHMED ABULSOROUR – SIVA VISVESWARAN, *Business process automation made easy with Java, Part 2*, http://www.javaworld.com/javaworld/jw-10-2002/jw-1018-process2_p.html

[28] XSL, http://www.w3.org/TR/xsl

[29] Apache XML FOP (Formatting Objects Project), http://xml.apache.org/fop

[30] XPath, http://www.w3.org/TR/xpath

[31] ANDRÉ TOST, *XML document processing in Java using XPath and XSLT*, in «JavaWorld», settembre 2000, http://www.javaworld.com/jw-09-2000/jw-0908-xpath.html

Capitolo 5

La Servlet API

GIOVANNI PULITI

Java è un linguaggio fortemente orientato alla programmazione delle reti e a quella distribuita; grazie alla facilità con cui questo linguaggio permette ad esempio di gestire le comunicazioni via socket e oggetti remoti per mezzo di RMI, è possibile realizzare applicazioni multistrato e distribuite in modo relativamente semplice.

Uno dei settori sul quale si focalizza maggiormente l'attenzione dello scenario della Information Technology è quello della programmazione web oriented, ovvero quella in cui la parte client è costituita da un semplice browser che interagisce con la parte server per mezzo del protocollo HTTP.

Questa tipologia di programmi, il cui modello di funzionamento viene tipicamente denominato Common Gateway Interface (CGI), ha l'obiettivo di permettere l'interfacciamento da parte di un client web con una serie di risorse e di servizi residenti sul server.

Sebbene CGI sia una definizione generica, con tale sigla si fa riferimento in genere a programmi scritti utilizzando linguaggi come il C o il PERL, ovvero secondo una filosofia di progettazione ormai superata.

Dal punto di vista della logica di esecuzione, questa è la sequenza delle operazioni:

- il client tramite browser effettua una chiamata al web server;
- il web server provvede a eseguire l'applicazione;
- l'applicazione dopo aver eseguito tutte le operazioni del caso, produce un output che viene passato al web server che lo invierà poi al client.

Poco prima dell'uscita definitiva del JDK 1.2, per la realizzazione di applicazioni CGI in Java, Sun ha introdotto la Servlet API, diventata in poco tempo una delle più importanti di tutta la piattaforma Java2.

Le servlet hanno introdotto alcune importanti innovazioni nel modello operativo del CGI, innovazioni che poi sono state adottate di riflesso sia nelle tecnologie più obsolete, sia in quelle direttamente concorrenti.

La API

Secondo la definizione ufficiale, una servlet è un "componente lato server per l'estensione di web server Java enabled", cioè un qualcosa di più generale del solo WWW. Una servlet è quindi un programma Java in esecuzione sul server e in grado di colloquiare con il client per mezzo del protocollo HTTP. Tipicamente questo si traduce nella possibilità di generare dinamicamente contenuti web da visualizzare nella finestra del client-browser.

I package che contengono tutte le classi necessarie per la programmazione delle servlet sono il javax.servlet e il javax.servlet.http: come si può intuire dal loro nome si tratta di Standard Extension API, ovvero non fanno parte del JDK core.

Per questo motivo, quando si lavora con le servlet, è importante specificare quale versione della API si utilizza, dato che il servlet engine utilizzato potrebbe non supportare l'ultima versione delle API. In questo capitolo, dove necessario, verranno affrontati i vari aspetti delle API indicando specificatamente ove siano presenti delle differenze fra le varie API: le versioni a cui si fa riferimento in questa sede sono la 2.0, 2.1, 2.2, 2.3 e 2.4 mentre si tralasceranno i dettagli relativi alla 1.0, trattandosi di una versione piuttosto vecchia non più supportata dalla maggior parte dei server.

Si tenga presente inoltre che, con la versione 2.2, le Servlet API sono entrate a far parte formalmente della cosiddetta Java 2 Enterprise Edition, al fianco di altre importanti tecnologie come JDBC, JNDI, JSP, EJB e RMI.

Questa nuova organizzazione logica, anche se non ha alterato l'architettura di base, ha introdotto alcune nuove definizioni e formalismi. L'innovazione più importante da questo punto di vista è l'introduzione del concetto di *container* al posto di *server*: un container è un oggetto all'interno del quale una servlet vive e nel quale trova un contesto di esecuzione personalizzato. Il concetto di container ha poi una visione più ampia dato che viene adottato anche nel caso di JSP ed EJB.

Soluzione full java

Essendo composta al 100% da classi Java, una servlet è integrabile con il resto della tecnologia Java dando luogo a soluzioni scalabili in maniera molto semplice.

Efficienza

Sebbene per quanto riguarda l'invocazione il meccanismo sia piuttosto simile al modello CGI, dal punto di vista del funzionamento si ha una situazione radicalmente diversa: ogni volta che il client esegue una richiesta di un servizio basato su CGI, il web server deve mandare in esecuzione un processo dedicato per quel client. Se *n* client effettuano la stessa richiesta, allora *n* processi devono essere prima istanziati e poi eseguiti. Questo comporta un notevole dispendio di tempo e

di risorse di macchina. Il modello multithread permette a una sola servlet, dopo la fase di inizializzazione, di servire un numero *n* di richieste senza la necessità di ulteriori esecuzioni. In questo caso infatti l'abbinamento container, servlet e classi Java caricate dinamicamente rappresenta un solo processo. Tutto questo porta a notevoli vantaggi dal punto di vista della efficienza e delle potenzialità operative.



In principio la API permetteva di implementare un modello non threaded, in cui una servlet poteva servire una richiesta per volta in modo sequenziale. L'interfaccia SingleThreadModel, utilizzata a questo scopo, è stata deprecata con le ultime versioni della API, a sottolineare che una servlet è per concezione un oggetto di front-end per la comunicazione multithreaded con lo strato client: eventuali sincronizzazioni devono essere effettuate negli strati sottostanti.

Migliore interfacciamento

Il ciclo di vita di una servlet prevede l'invocazione in callback da parte del container dei metodi init e service, cosa che permette non solo di ottimizzare le risorse ma anche di migliorare la modalità di interfacciamento con il client. Il primo infatti serve per inizializzare la servlet, ed è il posto dove tipicamente si eseguono le operazioni computazionalmente costose da effettuare una volta per tutte (come la connessione a un DB). Il metodo service invece è quello che viene mandato in esecuzione al momento di una chiamata POST o GET: il server offre una gestione tramite thread del metodo service(), ovvero per ogni client che effettui una richiesta di servizio, il server manderà in esecuzione un thread che eseguirà il metodo service o, in maniera equivalente, doGet() o doPost(). Il tutto in modo automatico e trasparente agli occhi del programmatore che dovrà solo preoccuparsi di scrivere il codice relativo alle operazioni da effettuare in funzione della richiesta di *un* client.

Leggerezza

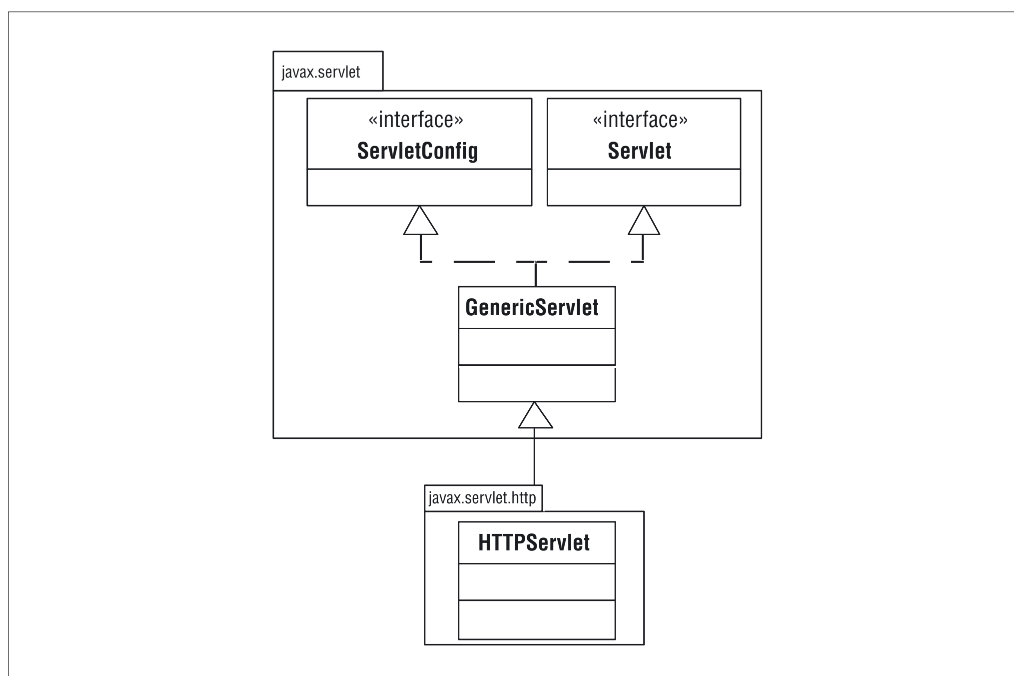
I due punti precedenti fanno capire come in realtà, oltre a una notevole semplicità di programmazione, il grosso vantaggio risieda nella leggerezza con cui il tutto viene gestito.

Portabilità

Dato che la maggior parte degli sviluppatori utilizza la piattaforma Windows per sviluppare e che le applicazioni web sono poi eseguite su sistemi Unix (Linux, Solaris, HP, Aix…) la portabilità di Java è in questo scenario quanto mai importante: nella specifica J2EE a tal proposito, con l'introduzione del concetto di container e deploy di una web application, questo porting è stato enormemente semplificato.

Portabilità di una applicazione web basata su servlet, pagine JSP o comunemente Java, significa anche che l'applicazione potrà essere spostata da una piattaforma a un'altra in funzione delle esigenze e del carico di lavoro: se per iniziare si è scelto di utilizzare un application server in esecuzione su Linux-Intel, con sforzo minimo il tutto potrà essere spostato su server station più

Figura 5.1 – *Gerarchia delle classi principali per la gestione delle Servlet.*



potenti a patto che vi sia una JVM per quel sistema; e praticamente tutti i sistemi operativi moderni hanno una JVM di ultima generazione.

Implementazione e ciclo di vita di una servlet

Per realizzare una servlet HTTP è sufficiente estendere la classe HttpServlet, appartenente al package javax.servlet.http, ridefinire il metodo init(), per personalizzare l'inizializzazione della servlet, ed i metodi service(), doPost() e doGet() per definirne invece il comportamento in funzione delle invocazioni del client (fig. 5.1).

La nascita: inizializzazione di una servlet

Il metodo init(), derivato dalla classe GenericServlet, ha lo scopo di effettuare tutte quelle operazioni necessarie per l'inizializzazione della servlet stessa, e per il corretto funzionamento successivo. La firma del metodo è la seguente

```
public void init(ServletConfig config) throws ServletException
```

Se durante il processo di inizializzazione si verifica un errore tale da compromettere il corretto funzionamento successivo, allora la servlet potrà segnalare tale evento generando una eccezione di tipo ServletException.

In questo caso, la servlet non verrà resa disponibile per l'invocazione e l'istanza appena creata verrà immediatamente rilasciata. Il metodo destroy(), in tale situazione, non verrà invocato dal server. Dopo il rilascio dell'istanza fallita, il server procederà immediatamente, alla istanziazione e inizializzazione di una nuova servlet; la generazione di una UnavailableException permette di specificare il tempo minimo necessario da attendere prima di intraprendere nuovamente il processo di inizializzazione. Il costruttore da utilizzare per creare questo tipo di eccezione è il seguente

```
public UnavailableException(java.lang.String msg, int seconds)
```

dove seconds è il tempo minimo di attesa; un valore negativo o pari a zero indica al server di istanziare una nuova servlet appena possibile.

Questo modo di gestire le anomalie lascia il controllo al servlet container e permette quindi di implementare strategie di backup: si deve perciò evitare l'invocazione del metodo System.exit().

Un caso tipico in cui può verificarsi il fallimento della inizializzazione di una servlet può essere quello in cui durante la init() si effettui una connessione con un database, utilizzata in seguito per inviare dati verso un client sotto forma di tabelle HTML. In questo caso potrebbe essere particolarmente utile imporre un periodo di attesa prima di effettuare una nuova connessione, ovvero prima di procedere alla creazione di un nuova servlet.

Nella ridefinizione di init(), quando si utilizza la versione 2.0 della API, la prima cosa da fare è invocare lo stesso metodo della classe padre.

```
public class myServlet {
...
  public void init(ServletConfig config) throws ServletException {
    super.init(config);
  }
  ...
}
```

Può apparire una operazione inutile, ma risulta essere particolarmente importante per permettere alla GenericServlet di effettuare altre operazioni di inizializzazione utilizzando il riferimento all'interfaccia ServletConfig passata come parametro.

Con la versione 2.1 della API, questa operazione non è più necessaria ed è possibile ridefinire direttamente il metodo init() senza parametri; è quindi possibile scrivere

```
public void init() throws ServletException {
  String UserId = getInitParameter("uid");
}
```

In questo caso il riferimento alla ServletConfig non viene perso, dato che la classe enericServletpermette questa apparente semplificazione. Il server al momento della inizializzazioneeffettuasempre una chiamata al metodo init(ServletConfig config): la differenza è che, con la versione 2.1 delle API, il GenericServlet effettua immediatamente la chiamata alla init senza parametri. Ad esempio

```
public class GenericServlet implements Servlet, ServletConfig {

  ServletConfig Config = null;

  public void init(ServletConfig config) throws ServletException {
    Config = config;
    log("init called");
    init();
  }

  public void init()   throws ServletException {
  }

  public String getInitParameter(String name) {
    return Config.getInitParameter(name);
  }

  // ecc...
}
```

Grazie al metodo getInitParameter(String), il quale effettua una invocazione direttamente al runtime su cui è in esecuzione la servlet, all'interno del metodo init() è possibile ricavare il valore dei parametri di inizializzazione.

Il metodo getParameterNames() permette di conoscere il nome dei parametri di inizializzazione.

Il resto della vita: i metodi doGet(), doPost() e service()

Dopo l'inizializzazione, una servlet si mette in attesa di una eventuale chiamata da parte del client, che potrà essere indistintamente una GET o una POST HTTP.

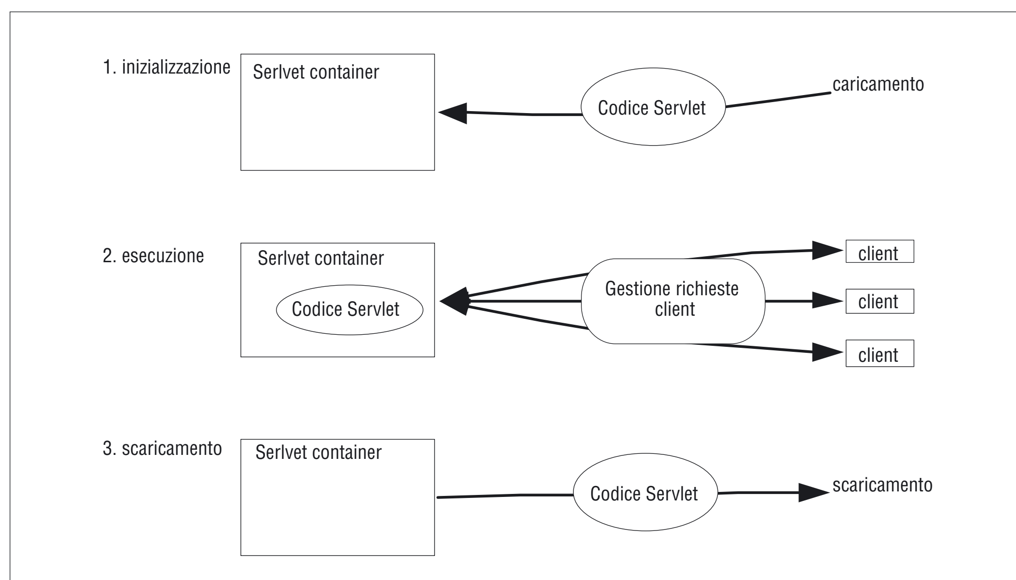
L'interfaccia Servlet mette a disposizione a questo scopo il metodo

```
public void service(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException
```

che viene invocato direttamente dal server in modalità multithread (il server per ogni invocazione da parte del client manda in esecuzione un metodo service in un thread separato).

Il service viene invocato indistintamente sia nel caso di una invocazione tipo GET che di una POST. La ridefinizione del metodo service permette di definire il comportamento della servlet

Figura 5.2 – *Ciclo di vita di una servlet. Si noti come per ogni client venga eseguito un thread separato.*



stessa. Se nella servlet è definito il metodo service(), esso è eseguito al posto dei metodi doGet() e doPost() (fig. 5.2).

I due parametri passati sono ServletRequest e ServletResponse, che permettono di interagire con la richiesta effettuata dal client e di inviare risposta tramite pacchetti HTTP. In seguito saranno analizzati in dettaglio gli aspetti legati a queste due interfacce.

Nel caso in cui invece si desideri implementare un controllo più fine si possono utilizzare i due metodi doGet() e doPost(), che risponderanno rispettivamente a una chiamata di tipo GET e a una di tipo POST.

Le firme dei metodi, molto simili a quella del metodo service(), sono:

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException
```

Con la API 2.2 è stato aggiunto all'interfaccia ServletConfig il metodo getServletName() che restituisce il nome con cui la servlet è stata registrata, o il nome della classe nel caso di una servlet non registrata.

Il codice che segue mostra come ricavare il nome della servlet per poter reperire ulteriori informazioni dal contesto corrente.

```
public void service(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {
  // invoca il metodo getServletName derivato
  // dalla classe padre GenericServlet
  String name = getServletName();
  // ricava il contesto
  ServletContext context = getServletContext();
  // ricava un attributo generico del servlet
  Object value = context.getAttribute(name + ".state");
}
```

La gestione delle eccezioni

La ServletException rappresenta l'eccezione tipicamente generata in tutti quei casi in cui uno dei metodi legati al ciclo di vita della servlet fallisca per qualche motivo. In base alla filosofia adottata nella gestione delle eccezioni si può quindi dire che la generica ServletException funziona come wrapper della particolare eccezione che di volta in volta verrà generata.

A volte una eccezione di questo tipo viene detta causa radice o, con un termine dal suono più familiare, *root cause*.

A questo scopo sono stati introdotti due nuovi costruttori della ServletException in grado di offrire una maggiore potenza descrittiva circa le cause dell'eventuale problema.

```
ServletException(Throwable rootCause)
ServletException(String message, Throwable rootCause)
```

Si supponga ad esempio di dover intercettare una InterruptedException e di propagare quindi una generica ServletException: utilizzando le Servlet API 2.0 si potrebbe scrivere

```
try {
  thread.sleep(100);
}
catch (InterruptedException e) {
  // si genera una eccezione generica
  // utilizzando esclusivamente il messaggio
  // testuale per specificare ulteriori dettagli
  throw new ServletException(e.getMessage());
}
```

che con la versione 2.1 potrebbe diventare

```
try {
  thread.sleep(100);
}
catch (InterruptedException e) {
  // si genera una eccezione specifica
  throw new ServletException(e);
```

```
}
```

In questo caso, dato che si propaga l'eccezione utilizzando un oggetto per la creazione della ServletException, il server riceve molte più informazioni rispetto a prima, dato che si utilizza un oggetto vero e proprio al posto di un semplice messaggio.

Questa nuova possibilità messa a disposizione dalle API 2.1 in effetti è utile, anche se un suo utilizzo comporta il rischio di dover riscrivere tutta la gestione delle eccezioni all'interno della stessa applicazione.

Per questo motivo, visto che l'aiuto fornito non è di importanza essenziale, si consiglia spesso di farne a meno.

Come interagire con una servlet: richieste e risposte

Una servlet può comunicare con il client in maniera bidirezionale per mezzo delle due interfacce HttpServletRequest e HttpServletResponse: la prima rappresenta la richiesta e contiene i dati provenienti dal client, mentre la seconda rappresenta la risposta e permette di incapsulare tutto ciò che deve essere inviato indietro al client stesso.

Domandare è lecito: HttpServletRequest

La HttpServletRequest, oltre a permettere l'accesso tutte le informazioni relative all'header http (come ad esempio i vari cookies memorizzati nella cache del browser), permette di ricavare i parametri passati insieme all'invocazione del client (fig 5.3).

Tali parametri sono inviati come di coppie nome–valore, sia che la richiesta sia di tipo GET che POST. Ogni parametro può assumere valori multipli.

L'interfaccia ServletRequest dalla quale discende la HttpServletRequest mette a disposizione alcuni metodi per ottenere i valori dei parametri passati alla servlet: ad esempio, per ottenere il valore di un parametro dato il nome, si può utilizzare il metodo fornito dalla interfaccia

```
public String getParameter(String name)
```

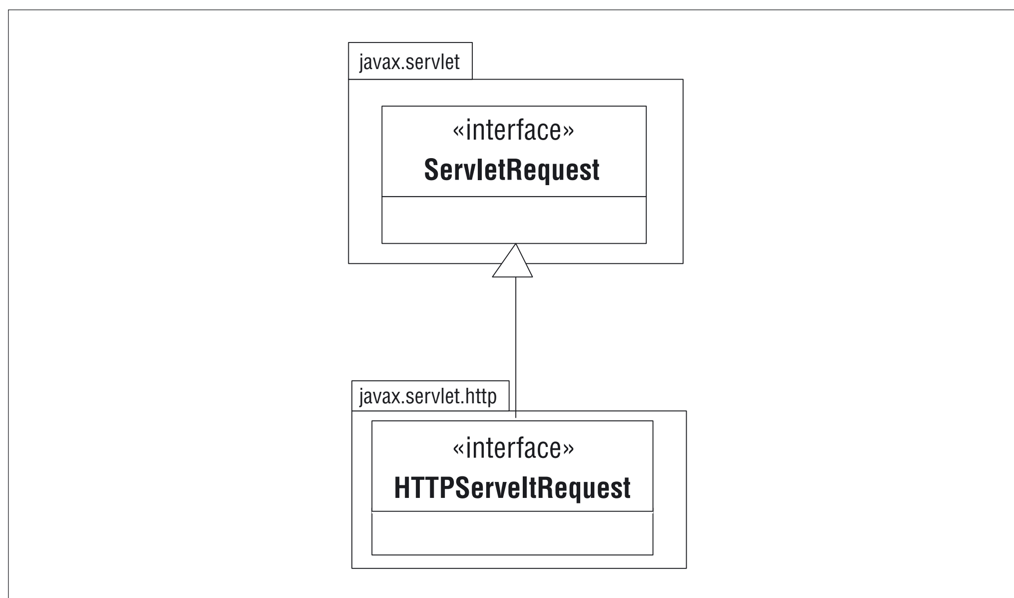
Nel caso di valori multipli per la stessa variabile — come quando l'invocazione viene fatta tramite un form HTML — fino alla API 2.0 non era specificato il formalismo con cui tali parametri dovevano essere restituiti dal server, il quale poteva restituire un array, una lista di stringhe separate da virgola o altro carattere.

Con la API 2.1 invece si ottiene sempre un array di stringhe, risultato analogo a quello fornito dal metodo

```
public String[] getParameterValues(String name)
```

Per ottenere invece tutti i nomi o tutti i valori dei parametri si possono utilizzare i metodi

Figura 5.3 – *Le classi utilizzabili per gestire le richieste da parte del client: ServletRequest e HttpServletRequest.*




```
public Enumeration getParameterNames()
public String[] getParameterValues(String name)
```

Infine, nel caso di una GET http, il metodo getQueryString() restituisce una stringa con tutti i parametri concatenati.

Nel caso in cui si attendano dati non strutturati, in forma testuale, e l'invocazione sia una POST, una PUT o una DELETE, si può utilizzare il metodo getReader(), che restituisce un BufferedReader.

Se invece i dati inviati sono in formato binario, allora è indicato utilizzare il metodo getInputStream(), il quale a sua volta restituisce un oggetto di tipo ServletInputStream.

Ecco di seguito un semplice esempio

```
public class MyServlet extends HttpServlet {
  public void service (HttpServletRequest req, HttpServletResponse res) throws ServletException {
    res.setContentType("text/html");
    res.setHeader("Pragma", "no-cache");
    res.writeHeaders();
    String param = req.getParameter("parametro");
    // esegue programma di servizio
    OutputStream out = res.getOutputStream();
    // usa out per scrivere sulla pagina di risposta
```

```
    }
}
```

Da notare l'utilizzo dello statement

```
OutputStream out = res.getOutputStream();
```

impiegato per ricavare lo stream di output con il quale inviare dati (ad esempio codice HTML) al client. I metodi

```
HttpServletRequest.getRealPath(String path)
ServletContext.getRealPath(String path)
```

possono essere utilizzati per determinare la collocazione sul file system per un dato URL: con la versione 2.1 della API, il primo di questi due metodi è stato deprecato, consentendo l'utilizzo solo della seconda versione. Il motivo di questa scelta è di praticità da un lato (ridurre la ridondanza) e di correttezza formale dall'altro: la collocazione di una risorsa sul file system è una informazione relativa al contesto di esecuzione della servlet non di richiesta.

Per quanto riguarda il formalismo da utilizzare per abbreviare un Uniform Resource Locator, con la versione 2.1 delle API è stata fatta chiarezza: se infatti nella versione precedente erano presenti metodi come `getRequestURL()` ma anche `encodeUrl()`, adesso la regola impone sempre l'utilizzo della lettera maiuscola. Ogni metodo contenente la versione in lettere minuscole è stato deprecato e rimpiazzato.

I metodi interessati sono

```
public boolean HttpServletRequest.isRequestedSessionIdFromUrl()

public String HttpServletRequest.encodeUrl(String url)
public String HttpServletRequest.encodeRedirectUrl(String url)
```

che, utilizzando per l'acronimo URL solo lettere maiuscole, diventano così

```
public boolean HttpServletRequest.isRequestedSessionIdFromURL()
public String HttpServletRequest.encodeURL(String url)
public String HttpServletRequest.encodeRedirectURL(String url)
```

Gli header associati alla chiamata

La codifica dei pacchetti HTTP prevede una parte di intestazione detta header dove si possono codificare le informazioni legate alla tipologia di trasmissione in atto.

Una servlet può accedere all'header della request HTTP per mezzo dei seguenti metodi della interfaccia `HttpServletRequest`:

```
public String getHeader(String name)
public Enumeration getHeaderNames()
```

Il getHeader() permette di accedere all'header a partire dal nome. Con la versione 2.2 della API è stato aggiunto anche il metodo

```
public Enumeration getHeaders(String name)
```

che permette di ricavare header multipli i quali possono essere presenti come nel caso del cache control.

In alcuni casi, gli header possono contenere informazioni di tipo non stringa, ovvero numerico o data: in questo caso può essere comodo utilizzare direttamente i metodi

```
public int getIntHeader(String name)
public long getDateHeader(String name)
```

che restituiscono direttamente il tipo più opportuno; nel caso che la conversione a tipo numerico non sia possibile, verrà generata una NumberFormatException, mentre se la conversione a data non è possibile, allora l'eccezione generata sarà una IllegalArgumentException.

Rispondere è cortesia: HttpServletResponse

La risposta di una servlet può essere inviata al client per mezzo di un oggetto di tipo HttpServletResponse, che offre due metodi per inviare dati al client: il getWriter() che restituisce un oggetto di tipo java.io.Writer, e il getOutputStream() che, come lascia intuire il nome, restituisce un oggetto di tipo ServletOutputStream (fig. 5.4).

Si deve utilizzare il primo tipo per inviare dati di tipo testuale al client, mentre un Writer può essere utile per inviare anche dati in forma binaria.

Da notare che la chiusura di un Writer o di un ServletOutputStream dopo l'invio dei dati permette al server di conoscere quando la risposta è completa.

Con i metodi setContentType() e setHeader() si può stabilire il tipo MIME della pagina di risposta e, nel caso sia HTML, avere un controllo fine su alcuni parametri.

Nell'esempio visto poco sopra, il content-type è stato impostato a "text/html", a indicare l'invio di una pagina HTML al browser, mentre "pragma no-cache" serve per forzare il browser a non memorizzare la pagina dalla cache (opzione questa molto importante nel caso di pagine a contenuto dinamico).

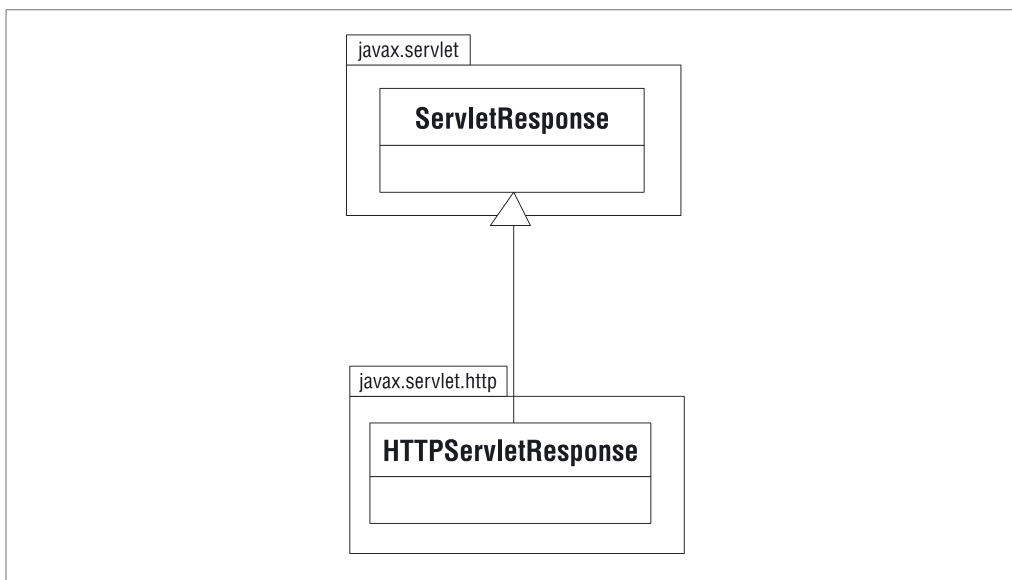
Gli aspetti legati alla modalità di interazione fra client e servlet, binaria o testuale, sono molto importanti al fine di comprendere il ruolo della Servlet API in relazione con le altre tecnologie offerte da Java, come ad esempio JSP.

Prima del rilascio della API 2.2, molti server implementavano tecniche di buffering per migliorare le prestazioni (utilizzando memorie tampone tipicamente di circa 8 kilobyte), mentre adesso il cosiddetto *response buffering* è parte integrante della specifica ufficiale.

Cinque nuovi metodi sono stati aggiunti all'interfaccia ServletResponse per la gestione diretta del buffer, sia che si utilizzi una comunicazione a mezzo di un ServletOutputStream sia di un Writer.

Il getBufferSize() restituisce la dimensione del buffer: nel caso in cui tale risultato sia un intero pari a zero, allora questo significa mancanza di buffering.

Figura 5.4 – *Le classi utilizzabili per la gestione delle risposte verso il client: ServletResponse e HTTPServletResponse.*



Per mezzo di setBufferSize () la servlet è in grado di impostare la dimensione del buffer associata a una risposta: in tal caso è sempre bene specificare una dimensione maggiore di quella utilizzata normalmente dalla servlet, al fine di riutilizzare la memoria già allocata e velocizzare ulteriormente la modifica. Inoltre deve essere invocato prima di ogni operazione di scrittura per mezzo di un ServletOutputStream o di un Writer.

Il metodo isCommitted() restituisce un valore booleano a indicare se siano stati inviati o meno dei dati verso il cliente. Il flushBuffer() invece forza l'invio dei dati presenti nel buffer.

Infine una chiamata delle reset() provoca la cancellazione di tutti i dati presenti nel buffer.

Se il buffer è stato appena vuotato a causa dell'invio della risposta verso il client, allora l'invocazione del metodo reset() provocherà la generazione di una eccezione di tipo IllegalStateException.

Una volta che il buffer viene riempito, il server dovrebbe immediatamente effettuare una flush del contenuto verso il client. Se si tratta della prima spedizione dati, si considera questo momento come l'inizio della commit.

```
public void doGet(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException {
  // si imposta un buffer di 8 kb
  res.setBufferSize(8 * 1024);
  res.setContentType("text/html");
  PrintWriter out = res.getWriter();
```

```
    // restituisce la dimensione del buffer
    // in questo caso 8192
    int size = res.getBufferSize();

    // si invia un messaggio nel buffer
    out.println("Messaggio inutile");
    // si ripulisce il buffer eliminando il
    // messaggio precedentemente scritto
    res.reset();

    // si scrive un altro messaggio nel buffer
    out.println("Altro messaggio inutile");

    // si ripulisce il buffer eliminando il
    // messaggio precedentemente scritto
    res.reset();

    // questo messaggio non appare se si invoca la sendError
    out.println("Messaggio probabilmente utile");

    // si esegue un controllo su un parametro importante
    if (req.getParameter("DBName") == null) {
      String MsgError = "Manca un parametro indispensabile: il nome del database ";
      res.sendError(res.SC_BAD_REQUEST, MsgError);
    }
  }
```

La gestione degli header nella risposta

Nell'inviare la risposta al client, la servlet può anche modificare l'header HTTP per mezzo dei metodi messi a disposizione dall'interfaccia HttpServletResponse: ad esempio il metodo

```
public void setHeader(String name, String value)
```

permette di impostare un header con nome e valore. Nel caso che uno o più valori con tale nome fossero già presenti, allora verrebbero tutti sostituiti con il nuovo valore.

Per aggiungere invece un valore a un nome specifico si può utilizzare il metodo

```
public void addHeader(String name, String value)
```

In alcuni casi i dati memorizzati nell'header HTTP possono essere gestiti meglio direttamente come dati numerici, o tipo data. In questo caso i metodi

```
public void addDateHeader(String name, long date)
public void setDateHeader(String name,long date)
public void addIntHeader(String name, int value)
```

```
public void setIntHeader(String name, int value)
```

permettono di effettuare in modo specifico le stesse operazioni di cui prima.

Al fine di inviare in modo corretto le informazioni memorizzate all'interno di un header, ogni operazione di modifica deve essere effettuata prima della commit della risposta, ovvero prima di inviare anche un solo byte verso il client.

Il metodo

```
public void sendRedirect(String location) throws IOException
```

imposta l'header e il body appropriati in modo da redirigere il client verso un URL differente. A tal punto, eventuali dati memorizzati nel buffer vengono immediatamente cancellati, dato che la trasmissione viene considerata completa.

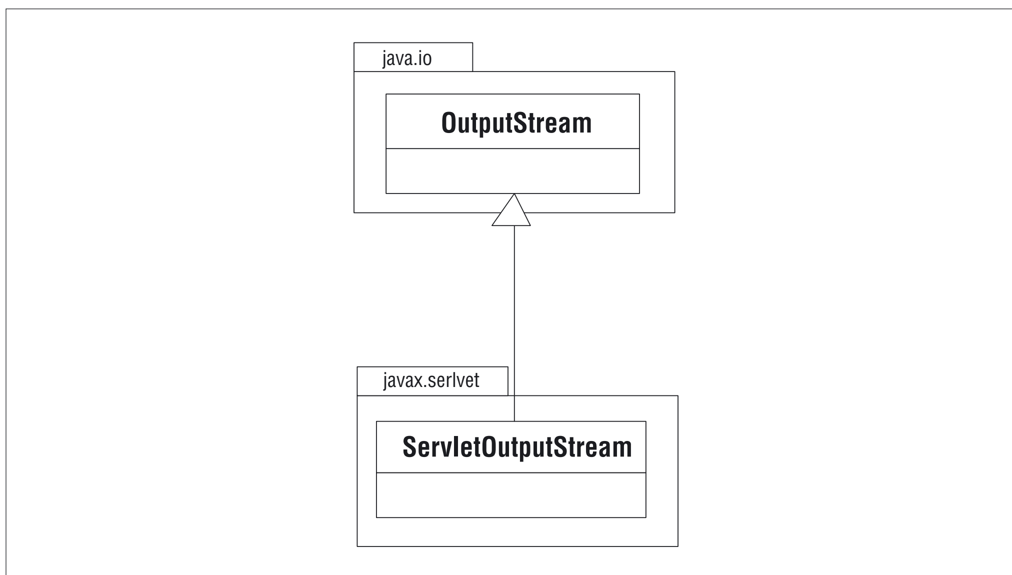
Il parametro passato può essere un URL relativo, anche se in tal caso il server dovrebbe sempre effettuare una trasformazione a URL assoluto. Nel caso in cui tale conversione non possa essere fatta per un qualsiasi motivo, allora verrà generata una eccezione di tipo `IllegalArgumentException`.

Anche l'invocazione del metodo `sendError()` forza il completamento della trasmissione: in questo caso tale metodo permette di specificare un parametro che verrà utilizzato come messaggio di errore nell'header stesso (fig. 5.5).

Analogamente si tenga presente che si ha la medesima situazione nel caso in cui sia stato inviato un quantitativo di informazioni pari a quanto specificato tramite `setContentLength()`.

Con la versione 2.1 della API il metodo

Figura 5.5 – *ServletOutputStream, lo stream associabile a una risposta.*



```
HttpServletResponse.setStatus(int sc, String sm)
```

è stato deprecato a favore dei metodi

```
HttpServletResponse.setStatus(int sc)
HttpServletResponse.sendError(String msg)
```

al fine di offrire maggiore eleganza e coerenza nella funzionalità dei metodi.

La fine della vita: distruzione di una servlet

A completamento del ciclo di vita, si trova la fase di distruzione della servlet, legata al metodo destroy(), il quale permette inoltre la terminazione del processo e il log dello status.

La ridefinizione di tale metodo, derivato dalla interfaccia Servlet, permette di specificare tutte le operazioni simmetriche alla inizializzazione, oltre a sincronizzare e rendere persistente lo stato della memoria.

Il codice che segue mostra quale sia il modo corretto di operare

```
public class myServlet extends HttpServlet {
  public void init(ServletConfig config) throws ServletException {
    // effettua operazioni di inizializzazione
    // per esempio l'apertura della connessione verso il db
  }

  public void destroy() {
    // effettua la chiusura della connessione
  }
}
```

È importante notare che la chiamata della destroy(), così come quella della init() è a carico del server che ha in gestione la servlet.

La distruzione di una servlet e il relativo scaricamento dalla memoria avviene solamente nel momento in cui tutte le chiamate a service() da parte dei client sono state eseguite.

Nel caso in cui tale metodo effettui una computazione più lunga del previsto, il server può dare atto alla distruzione della servlet anche se il metodo non ha terminato. In questo caso è compito del programmatore adottare opportune tecniche per lo sviluppo della servlet, cosa che sarà vista in seguito.

Anche se non strettamente legato al ciclo di vita di una servlet, occorre conoscere il metodo getServletInfo() che di solito viene annoverato fra i metodi principali, accanto a init() e service(). Il suo scopo è quello di fornire un messaggio descrittivo sul funzionamento della servlet, messaggio che viene in genere visualizzato nella finestra di gestione delle servlet del container.

Nel caso in cui si desideri personalizzare una servlet con un messaggio descrittivo, si può ridefinire il metodo getServletInfo(), come ad esempio nella parte di codice che segue

```
public String getServletInfo() {
  return "MyServlet – Una servlet personalizzata – V 1.0"
}
```

L'habitat di una servlet: il servlet context

Il servlet context rappresenta l'ambiente di esecuzione all'interno del quale una servlet viene fatta eseguire. L'interfaccia ServletContext mette a disposizione una serie di metodi piuttosto utili per interagire con il contesto della applicazione, o per accedere alle risorse messe a disposizione della servlet stessa.

Ad esempio una servlet, utilizzando un oggetto di questo tipo, può effettuare il log di tutti gli eventi generati, ottenere l'URL associato a tutte le risorse messe a disposizione, e modificare gli attributi ai quali la servlet può accedere.

Il server su cui la servlet è in esecuzione è responsabile di fornire una implementazione della interfaccia ServletContext, e permette l'utilizzo di una serie di parametri di inizializzazione, che verranno utilizzati dalla servlet all'interno del metodo init().

Con la versione 2.1 della API sono state effettuate una serie di modifiche atte a limitare il legame con l'implementazione o il funzionamento del server sottostante: lo scopo è di garantire un più alto livello di sicurezza e correttezza di funzionamento.

Sicuramente una delle modifiche più importanti in tal senso riguarda il metodo

```
SerlvetContext.getServlet(Sring ServletName)
```

il quale permette di ottenere un riferimento a una servlet correntemente installata nel server e quindi di invocarne i metodi pubblici: per fare questo è necessario conoscerne il nome, ottenerne un riferimento all'oggetto Servlet e infine invocarne i metodi pubblici. Ad esempio si potrebbe pensare di scrivere

```
public class myServlet1 extends HttpServlet {

public void doGet (HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {
    myServlet2 serv2;
    ServletConfig Config = this.getServletConfig();
    ServletContext Context = Config.getServletContext();
    serv2 = (myServlet2) Context.getServlet("servlet2");
    serv2.dosomething();
    serv2.doGet(req, res);
  }
}
```

dove myServlet2 è la classe della servlet che si vuole invocare, e servlet2 è invece il nome logico con cui tale servlet è stata installata nel container.

Nella la nuova API il metodo getServlet() restituisce null poiché il suo utilizzo potrebbe dar vita a scenari inconsistenti nel caso in cui il server abbia invocato il metodo destroy() della servlet: tale metodo infatti è invocabile in ogni momento senza la possibilità o necessità di generare messaggi di avvertimento.

Inoltre, nel caso in cui il server utilizzi tecniche di load balancing grazie all'uso di JVM distribuite, non è possibile sapere in un determinato istante dove la servlet sia fisicamente in esecuzione.

Sun ha da sempre dichiarato come pericoloso l'utilizzo di tale metodo: la sua presenza si era resa necessaria per permettere la condivisione di informazioni e lo scambio di messaggi fra servlet differenti, cosa questa che prende il nome di *inter servlet communication*: tale obiettivo potrebbe essere messo in pratica utilizzando ad esempio il pattern Singleton oppure facendo uso di file condivisi.

Una soluzione più elegante e coerente con il resto dell'architettura consiste nell'utilizzare il ServletContext come repository di oggetti comuni (attributi di contesto), in modo da condividere informazioni fra servlet che sono in esecuzione nello stesso contesto: a tale scopo sono stati aggiunti alla ServletContext una serie di metodi, fra cui

```
public void setAttribute(String name, Object o)
public Object getAttribute(String name)
```

che permettono di impostare e ricavare un attributo dato il suo nome. Il secondo era presente fino dalla versione 1.0 delle API, anche se permetteva solamente di ricavare nel server attributi a sola lettura inseriti dal costruttore.

La gestione degli attributi di contesto è molto importante, dato che permette tra le altre cose la comunicazione della servlet con altri componenti in esecuzione nel container, come ad esempio le pagine JSP.

Completano la lista i metodi

```
public Enumeration getAttributeNames()
public void removeAttribute(String name)
```

i quali consentono rispettivamente di ottenere la lista di tutti i riferimenti, o di rimuoverne uno dato il nome.

Ogni attributo può avere un solo valore; gli attributi che iniziano per java. e javax. sono riservati, così come quelli che iniziano per sun. e com.sun. Sun, per la definizione degli attributi, suggerisce di seguire la regola del dominio rovesciato, adottata anche per la denominazione dei package.

La comunicazione fra servlet per mezzo del contesto comune è di grosso aiuto nel caso si debba effettuare dinamicamente la distribuzione delle servlet fra più JVM, offrendo in tal modo un sistema di comunicazione nativo funzionante anche nel caso in cui la struttura sottostante (le varie JVM) venga modificata.

Nei servlet container che seguono le specifiche più vecchie, tutte le servlet sono collocate nel medesimo ServletContext, dato che non vi è differenza fra un contesto e un altro; con la nascita del concetto di web application e con il crescere delle applicazioni stesse, inizia ad avere senso il raggruppamento e la conseguente suddivisione.

Per poter permettere la comunicazione fra contesti differenti, è stato introdotto il metodo

```
ServletContext.getContext(String urlpath)
```

il quale restituisce un ServletContext per un dato URL fornito come parametro.

L'interazione con gli altri oggetti del contesto

Si è già potuto vedere come l'utilizzo degli attributi di contesto permetta l'interazione fra i vari elementi appartenenti allo stesso ServletContext.

Con l'avvento della versione 2.1 della Servlet API e ancor più con la versione 2.2, è stata introdotta la possibilità non solo di scambiare messaggi, ma anche di propagare richieste e risposte fra le varie servlet dello stesso contesto.

Una delle innovazioni più interessanti introdotte (con la API 2.1) è la possibilità di delegare la gestione di una richiesta a un altro componente in esecuzione sul server.

Una servlet può quindi

- inoltrare una richiesta a un altro componente o a un'altra servlet dopo che ne abbia effettuata una prima elaborazione preliminare: questo è uno scenario tipico utilizzato con le JSP;
- includere nella propria risposta quella derivante dalla computazione di un componente esterno.

Queste due operazioni, che rientrano sotto il termine di *request delegation*, permettono di implementare in modo più organico la cosiddetta programmazione server side.

La API 2.1 fornisce una nuova interfaccia, la RequestDispatcher, ottenibile per mezzo del metodo

```
ServletContext.getRequestDispatcher(String path)
```

Il reference ottenuto permette di inoltrare la request direttamente all'oggetto identificato dal path passato come parametro. La RequestDispatcher fornisce i due metodi

```
public void forward(ServletRequest request, ServletResponse response) throws ServletException, IOException
public void include(ServletRequest request, ServletResponse response) throws ServletException, IOException
```

Il primo permette di inoltrare una richiesta pervenuta: al fine di garantire che il controllo passi totalmente al secondo oggetto, è necessario che l'invocazione della forward() avvenga prima dell'ottenimento di un ServletOutputStream o della creazione di un Writer da utilizzare per l'invio dei dati verso il client.

La chiamata alla include() invece può essere fatta in ogni momento, dato che il controllo resta alla servlet chiamante.

Ecco una breve porzione di codice in cui viene mostrato come includere l'output di un'altra servlet

```
// mostra un primo output di preparazione
out.println("Ecco la risposta della interrogazione:");

RequestDispatcher dispatcher = getServletContext();
dispatcher.getRequestDispatcher("/servlet/DBServlet?param=value");
dispatcher.include(req, res);
out.println("Si desidera effettuare una nuova ricerca?");
```

In questo caso la prima servlet si serve di una seconda, DBservlet, per effettuare le ricerche in un database relazionale. DBservlet emette una risposta automaticamente inviata al client, che riceve successivamente anche la risposta della prima servlet.

Invece di invocare DBservlet per mezzo della chiamata all'URL passando i parametri di esecuzione, per inoltrare una richiesta si possono impostare manualmente gli attributi della ServletRequest per mezzo del metodo ServletRequest.setAttribute().

Ad esempio

```
// mostra un primo output di preparazione
RequestDispatcher dispatcher = getServletContext();
out.println("Ecco le possibili risposte");
dispatcher.getRequestDispatcher("/servlet/DBServlet");

out.println("La prima risposta ");
// effettua una prima invocazione
String value1 = ...
req.setAttribute("param", value1);
dispatcher.include(req, res);

out.println("La seconda risposta ");
// effettua una seconda invocazione
String value2 = ...
req.setAttribute("param", value2);
dispatcher.include(req, res);

out.println("Si desidera effettuare una nuova ricerca?");
```

In questo caso la servlet chiamata può ricavare il valore dell'attributo legato alla invocazione per mezzo di una istruzione del tipo

```
String value1 = req.getAttribute("param");
```

o , in maniera equivalente, per mezzo di una più generica

```
String[] values = req.getAttributeNames();
```

Il vantaggio di utilizzare gli attributi invece delle semplici query string è che permette di passare come parametri oggetti al posto di semplici stringhe.

Con la API 2.2 è stato anche aggiunto alla interfaccia ServletContext il metodo

```
public RequestDispatcher getNamedDispatcher(java.lang.String name)
```

che permette di inoltrare una richiesta a un componente specificato per mezzo del suo nome univoco, piuttosto che utilizzando un URL. È questa una funzionalità resa possibile grazie al concetto di container e permette di inoltrare le richieste verso oggetti che non sono resi pubblici tramite un indirizzo.

Nella interfaccia ServletRequest, il metodo

```
public RequestDispatcher getRequestDispatcher(String path)
```

è stato modificato in maniera da accettare un URL relativo come parametro, differentemente dal getRequestDispatcher() il quale invece accetta solamente URL completi.

Nel caso di web application, l'URL relativo può essere utilizzato, ad esempio, per inoltrare richieste a oggetti facenti parte dello stesso contesto.

Anche il metodo

```
public void sendRedirect(String location)
```

è stato modificato in modo da accettare URL relativi, che verranno anche in questo caso modificati direttamente dal servlet container aggiungendo l'indirizzo URL del contesto dove viene eseguita la servlet.

Il context e le applicazioni distribuite

Sempre con la API 2.2 è stato ridefinito e standardizzato il funzionamento del server in modo da permettere il passaggio di oggetti condivisi fra VM distribuite, senza la generazione di indesiderate ClassCastException. Questo inconveniente poteva generarsi quando servlet differenti venivano caricate da classloader diversi, dato che due oggetti caricati in memoria da due loader diversi non possono essere "castati" l'un l'altro.

In base a tale regola, la modifica effettuata su una servlet, oltre che implicare il ricaricamento in memoria della servlet stessa, provocherà anche il reload di tutte le classi da essa utilizzate: questo da un lato può comportare una diminuzione delle prestazioni, ma ha il grosso pregio di permettere la sincronizzazione del codice e di semplificare non poco la fase di sviluppo delle applicazioni.

Infine si tenga presente che la specifica 2.2 ribadisce che, nel caso di applicazioni distribuite, un contesto debba essere utilizzato su una sola JVM: i context attribute non possono essere quindi utilizzati per memorizzare attributi a scope globale. Questo obiettivo può essere perseguito utilizzando risorse esterne al server, come ad esempio database o componenti EJB.

Una applicazione marcata come distribuibile segue inoltre altre importanti regole per le sessioni utente: i server infatti utilizzano la cosiddetta *session affinity* per una gestione efficace del mantenimento dello stato fra i vari server. Questo implica che tutte le richieste a una singola

sessione da parte di un certo utente sono gestite da una singola JVM alla volta, cosa che tra l'altro evita di dover sincronizzare fra i vari server le informazioni memorizzate nella sessione.

Visto che il meccanismo utilizzato per trasferire i dati è al solito quello della serializzazione, le variabili memorizzate nelle varie sessioni devono essere serializzabili. Nel caso in cui un oggetto non sia serializzabile, il server produrrà una IllegalArgumentException.

Resource abstraction

Con la versione 2.1 della API è stato formalizzato il concetto di astrazione delle risorse, consentendo così l'accesso alle risorse di sistema indipendentemente dalla loro collocazione nel sistema, essendo i vari riferimenti gestiti sotto forma di URL.

Questo permette di gestire una servlet come oggetto indipendente dal contesto e di spostarla da un server a un altro senza particolari difficoltà o senza la necessità di dover riscrivere buona parte del codice.

Il metodo che permette di ottenere una risorsa è il

```
ServletContext.getResource(String uripath)
```

dove uripath rappresenta l'URI in cui è collocata tale risorsa: è compito del web server effettuare il mapping fra risorsa vera e propria e URL.

Una risorsa astratta non può essere una risorsa attiva (servlet, CGI, JSP, programmi o simili), per le quali si deve utilizzare la RequestDispatcher. Ad esempio, una invocazione getResource("/index.jsp") restituisce il sorgente del file JSP e non il risultato dell'elaborazione di tale file da parte del container.

Una volta ottenuto il riferimento all'URL della risorsa è possibile effettuare le operazioni permesse su tale risorsa: ad esempio si può pensare di individuare una pagina HTML e di effettuarne successivamente la stampa verso il client.

```
String res="/static_html/header.html";
URL url = getServletContext().getResource(res);
out.println(url.getContent());
```

Qui il file header.html rappresenta la prima parte della pagina HTML da inviare al client, e può essere collocato sotto la web root sul container dove viene eseguita la servlet, o in alternativa su un server remoto.

Una risorsa astratta possiede una serie di caratteristiche che possono essere conosciute utilizzando una connessione verso l'URL relativo alla risorsa; ad esempio

```
// ricava l'url della front page
URL url = getServletContext().getResource("/");
URLConnection con = url.openConnection();
con.connect();
int ContentLength = con.getContentLength();
```

```
String ContentType = con.getContentType();
long Expiration = con.getExpiration();
long LastModified = con.getLastModified();
```

Il metodo

```
InputStream in = getServletContext().getResourceAsStream("/")
```

permette di ottenere direttamente uno stream associato alla risorsa remota, senza doverlo aprire esplicitamente, come ad esempio

```
URL url = getServletContext().getResource("/");
InputStream in = url.openStream();
```

Nel caso in cui la risorsa lo permetta, è possibile utilizzare il metodo getResource() per ottenere un riferimento in scrittura su una risorsa; ad esempio

```
URL url = getServletContext().getResource("/myservlet.log");
URLConnection con = url.openConnection();
con.setDoOutput(true);
OutputStream out = con.getOutputStream();
PrintWriter pw = new PrintWriter(new OutputStreamWriter(out));
pw.println("Evento verificatosi il" + (new Date());
pw.close();
out.close();
```

Nel caso non si possa o non si voglia utilizzare esclusivamente la API 2.1, si potrà continuare a effettuare il riferimento alle risorse di tipo file utilizzando i metodi standard per l'accesso a file, come ad esempio getPathTranslated(), anche se in questo caso un oggetto di tipo File è strettamente legato alla macchina e non è portabile.

Con la versione 2.2 delle API è stato aggiunto al ServletContext l'attributo

```
javax.servlet.context.tempdir
```

che contiene il nome della directory temporanea utilizzata dal server: tale informazione può essere utilizzata, tramite un oggetto di tipo java.io.File, per effettuare operazioni di I/O nella directory temporanea di cui sopra. Ecco un esempio

```
public void doGet(HttpServletRequest req, HttpServletResponse res) throws ServletException,IOException{

  // ricava la directory temporanea come un object File
  String Attr="javax.servlet.context.tempdir";
  Object o = getServletContext().getAttribute(Attr);
  File dir = (File) o;
```

```
        // Crea un file temporaneo nella temp dir
        File f = File.createTempFile("xxx", ".tmp", dir);

        // inzia le operazioni di scrittura
        FileOutputStream fos = new FileOutputStream(f);

    }
```

Sia durante la fase di debug che durante la normale esecuzione, può essere molto utile controllare il funzionamento della servlet ed eventualmente effettuare il log delle operazioni eseguite. È per questo motivo che il ServletContext, per effettuare il salvataggio di messaggi di log, mette a disposizione il metodo

```
log(Exception e, String msg)
```

(disponibile fino alla versione 2.1 della Servlet API), con il quale è possibile memorizzare un messaggio relativo a una generica Exception. Con il passaggio alla versione 2.1 della API, tale metodo è stato deprecato, e sostituito con

```
log(String msg, Throwable e)
```

Questo cambiamento, oltre a essere maggiormente in linea con lo standard Java che vuole il posizionamento delle eccezioni in ultima posizione fra i vari parametri, consente l'utilizzo di una eccezione più generica della semplice Exception.

Inoltre il metodo è stato spostato nella classe GenericServlet, cosa che permette una sua invocazione del log senza dover prima ricavare un riferimento al ServletContext.

Il context e la versione della API utilizzata

Dato che l'evoluzione della Servlet API — pur dando vita, di versione in versione, a un framework sempre più robusto, potente e coerente — introduce non pochi problemi di compatibilità, sebbene le variazioni in genere garantiscano la retrocompatibilità, può essere utile in certi casi ricavare la versione della API utilizzata: dalla 2.1 sono disponibili i seguenti metodi che restituiscono questo genere di informazioni

```
public int getMajorVersion()
public int getMinorVersion()
```

i quali, nel caso della versione 2.1 restituiscono rispettivamente i due interi 2 e 1.

Servlet e internazionalizzazione

La possibilità di rendere un'applicazione sensibile alla localizzazione geografica in cui viene eseguita è sicuramente una delle caratteristiche più interessanti che Java mette a disposizione grazie alla internazionalizzazione.

Nel caso della programmazione web questa caratteristica è sicuramente ancor più importante, dato che un client potrebbe essere ubicato in una qualsiasi regione del mondo connessa alla rete.

È per questo motivo che, oltre al normale supporto per la localizzazione nel caso della Servlet API 2.2 sono stati aggiunti alcuni metodi alla interfaccia HttpServletRequest con lo scopo di determinare la localizzazione del client e quindi agire di conseguenza.

Il metodo getLocale() restituisce un oggetto di tipo java.util.Locale determinato in base alle informazioni memorizzate nell'header Accept-Language della richiesta.

Il metodo getLocales() invece restituisce la lista di tutti i Locale accettati dal client con quello preferito in testa alla lista.

Parallelamente, il metodo setLocale() della HttpServletResponse permette di impostare il Locale opportuno per inviare la risposta nel modo migliore verso il client.

Ad esempio questi metodi potrebbero essere utilizzati nel seguente modo

```
public void doGet(HttpServletRequest req,
HttpServletResponse res) throws ServletException,IOException {

   res.setContentType("text/html");
   Locale locale = req.getLocale();
   res.setLocale(locale);
   PrintWriter out = res.getWriter();

   // scrive un output sulla base della lingua
   // ricavata tramite locale.getLanguage()
}
```

Invocazione della servlet

Come si è potuto comprendere nella sezione dedicata al ciclo di vita delle servlet, esse "vivono" solamente durante l'invocazione di una GET o di una POST HTTP: è quindi importante capire come sia possibile effettuare una invocazione con parametri da parte del client.

Di seguito saranno quindi prese in esame le principali modalità di invocazione, che devono essere aggiunte al caso di invocazione di servlet da servlet.

Invocazione diretta dell'URL

Il modo più semplice per invocare una servlet è quello di invocare l'URL corrispondente alla servlet, ad esempio tramite browser. Questo tipo di invocazione corrisponde a una GET HTTP ed ha la seguente sintassi

```
http://nomehost:porta/nomeservlet[?parametro1=valore1&parametro2=valore2]
```

ad esempio

```
http://www.mysite.com/myservice?url=myurl&user=franco
```

In questo caso i parametri sono passati direttamente alla servlet per mezzo della sintassi particolare dell'invocazione della servlet.

Il container potrebbe aver abilitato un alias con nome simbolico, per cui l'invocazione potrebbe prendere la forma

```
http://www.mysite.com/myservice.html?url=myurl&user=franco
```

Indipendentemente dalla modalità di invocazione, il codice necessario per ricavare un parametro passato dall'interno del codice Java è sempre lo stesso, ovvero per mezzo del metodo getParameter().

Un'invocazione di tipo GET corrisponde a una comunicazione di messaggi di testo della lunghezza massima di 256 caratteri, che diventa la lunghezza massima della stringa da passare come parametro alla servlet.

Invocazione per mezzo di un form HTML

Tramite la realizzazione di un form HTML è possibile invocare una servlet specificandone l'URL associato. In questo caso oltre ai parametri da passare alla servlet si può indicare la tipologia di invocazione (GET o POST).

Ad esempio un form HTML potrebbe essere

```
<FORM METHOD="POST" ACTION=="http://www.mokabyte.it/servlet/MyServlet">
     Immetti un testo: <BR> <INPUT TYPE="text" NAME="data"><BR>
   <input type="submit" name="B1" value="ok">
 </FORM>
```

che dà luogo a un form con una casella di testo (il cui contenuto poi verrà inviato alla servlet sotto forma di parametro denominato in questo caso *data*), e di un pulsante con etichetta "OK" (fig. 5.6).

In questo caso la servlet viene invocata effettivamente alla pressione del pulsante del form: il browser provvede a raccogliere tutti i dati e a inviarli alla servlet il cui URL è specificato per mezzo del tag ACTION.

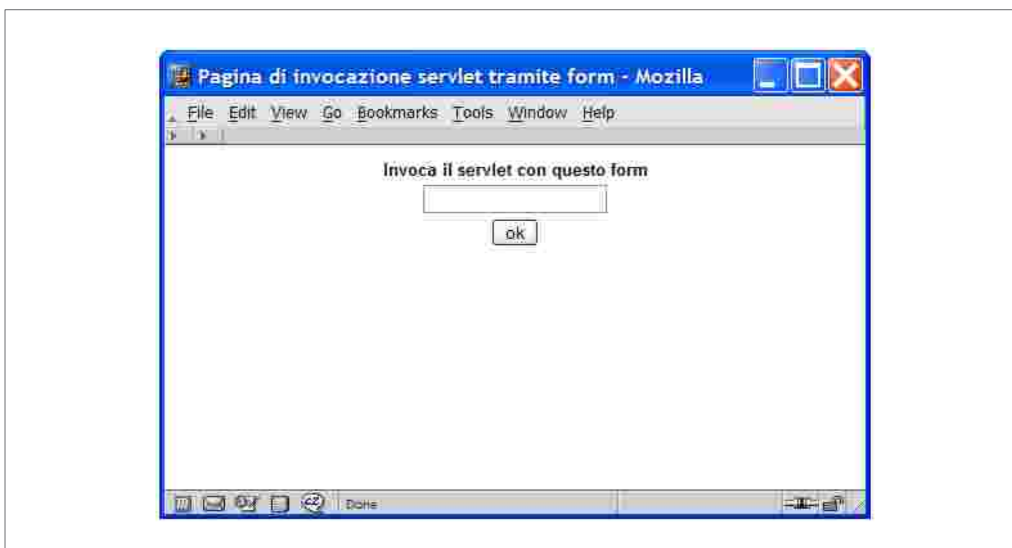
L'esempio che segue mostra come sia possibile ricavare tutti i parametri passati alla servlet per mezzo di una invocazione tramite form.

```
import java.io.*;
import java.util.*;

import javax.servlet.*;
import javax.servlet.http.*;


public class SimpleFormServlet extends HttpServlet {
```

Figura 5.6 – *Un tipico form HTML per l'invocazione di una servlet.*




```
public void service(HttpServletRequest req, HttpServletResponse res) throws IOException {

  Enumeration keys;
  String key;
  String value;
  String title;

  res.setContentType("text/html");
  ServletOutputStream out = res.getOutputStream();
  out.println("<HEAD><TITLE>");
  out.println("SimpleFormServletOutput");
  out.println("</TITLE></HEAD><BODY>");

  out.println("<h1> Questo è l'output del servlet");
  out.println("SimpleFormServlet </h1>");

  // ricavo i nomi e i valori dei parametri
  keys = req.getParameterNames();
  while (keys.hasMoreElements()) {
    key = (String) keys.nextElement();
    value = req.getParameter(key);
```

```
        out.println("<P><B>");
        out.print("Nome Parametro: </B>");
        out.print(key);
        out.print("<BR> <B> valore:</B>");
        out.print(value);
      }
      out.println("</BODY>");
    }

    public String getServletInfo() {
      String msg = "SimpleFormServlet"
      msg = msg + "ricava la lista di tutti i parametri inviati ";
      msg = msg + "tramite form ed invia la lista client ";
      return msg;
    }

  }
```

In questo caso i parametri sono ricavati tutti, e tramite pagine HTML viene inviata una risposta al client contenente tutti i nomi e i valori passati.

Il codice HTML della pagina che invoca tale servlet è piuttosto banale: ad esempio si potrebbe avere

```
<form METHOD=POST ACTION="http://www.mokabyte.it/form">
<input TYPE=hidden SIZE=30 NAME="nascosto" VALUE="messaggio segreto">
<table>
<tr>
<td>Nome</td>
<td><input TYPE=text SIZE=30 NAME="nome"></td>
</tr>

<tr>
<td>Cognome</td>
<td><input TYPE=text SIZE=30 NAME="cognome"></td>
</tr>

<tr>
<td>Indirizzo</td>
<td><input TYPE=text SIZE=30 NAME="indirizzo"></td>
</tr>

...

</table>
<p><input TYPE="submit" VALUE=" ok "><input TYPE="reset" VALUE="Clear Form"></center>
```

```
</form>
```

Si noti la presenza di un parametro nascosto, così come di due tag input con TYPE uguale a "submit" (è il pulsante che attiva l'invocazione), e di "reset" (il pulsante che pulisce il form). Infine si noti che l'action associata a tale form corrisponde a una serlvet *mappata* con nome logico form.



L'utilizzo di un form HTML non è altro che meccanismo per semplificare l'invocazione e l'organizzazione dei parametri. Dal punto di vista della servlet questa modalità di invocazione è del tutto analoga alle altre.

Il tag SERVLET

Un modo diverso di utilizzare le servlet per creare pagine HTML in modo dinamico è quello basato sul tag SERVLET che deve essere inserito all'interno del codice HTML. In questo caso la pagina, che si chiama server side e che deve avere estensione .shtml, viene preprocessata prima di essere inviata al client: in tale fase il codice HTML contenuto all'interno della coppia <SERVLET> ... </SERVLET> viene sostituito con l'output prodotto dalla servlet.

Il web server deve poter supportare il tag SERVLET (possibilità in genere specificata nelle caratteristiche del prodotto), altrimenti la pagina non verrà modificata, inviando al client il codice HTML originario.

Con l'introduzione delle tecnologia JSP, che può esserne considerata l'evoluzione, tale tecnica non viene più utilizzata, sia per la limitazione e la minor potenza espressiva rispetto a JSP, sia, soprattutto, perché è poco efficiente: per ogni invocazione da parte del client infatti il server deve elaborare la pagina HTML e inserire il risultato della servlet, senza la possibilità di eseguire alcuna operazione di compilazione o memorizzazione come avviene invece in JSP. Un esempio di pagina HTML server side potrebbe essere

```
<HTML>
<BODY>

<!— parte grafica HTML... -->
<p>The response is: <i>
  <SERVLET NAME=MyServlet>
  <PARAM NAME=param1 VALUE=val1>
  <PARAM NAME=param2 VALUE=val2>
  </SERVLET>
</i></p>
<!— parte grafica HTML... -->
</BODY>
</HTML>
```

Per quanto riguarda l'implementazione della servlet, non ci sono particolari differenze rispetto agli altri casi, se non per il fatto che la servlet deve inviare obbligatoriamente una risposta al client.

Invocazione nelle web application: serlvetMapping

I meccanismi di invocazione di una servlet visti fino a ora si basano essenzialmente sulla invocazione di un particolare URL su cui poi viene mappata una servlet. Tale mapping viene definito all'interno del file web.xml contenuto all'interno della web application che contiene la servlet.

Con l'introduzione del concetto di web application e di deploy all'interno di un container, tale file rappresenta il punto centrale che rende omogenea la procedura di configurazione di servlet, pagine JSP, filtri e listener di contesto. Tutte le volte quindi che un client (browser, applicazione standalone, applicazione embedded in un telefonino, …) effettua l'invocazione di un URL su cui è mappata una servlet, in realtà il controllo viene passato alla servlet perché nel file web.xml vi è un pezzo di script XML del tipo:

```
<servlet-mapping>
  <servlet-name>MyServlet</servlet-name>
  <url-pattern>*.servlet</url-pattern>
</servlet-mapping>
```

In questo caso tutte le invocazioni che abbiano come ultima parte dell'URL la parola "servlet" verranno redirette alla servlet MyServlet.

Terminazione corretta delle servlet

Può accadere che i metodi di servizio, ovvero quelli che rispondono alle invocazioni del client, primo fra tutti il service(), non abbiano terminato la loro attività al momento in cui il server invoca destroy().

Tecnicamente questa situazione prende il nome di *potentially long-running service requests* e deve essere opportunamente gestita tramite alcune operazioni, fra cui:

- tenere traccia del numero di thread correntemente in esecuzione durante una chiamata al metodo service();
- implementare il cosiddetto "shutdown pulito", permettendo al metodo destroy() della servlet di notificare lo shutdown in atto ai vari thread, cosentendo loro di finire il lavoro;
- eventualmente effettuare dei controlli nei vari punti a maggiore carico computazionale, in modo da simulare uno shutdown, per verificare la costante responsività dei thread a lunga computazione.

Per tenere traccia delle richieste da parte dei client, si può ad esempio utilizzare una variabile nella servlet che svolga proprio la funzione di contatore. Questa variabile deve essere accessibile sia per la lettura che per la scrittura, attraverso metodi sincronizzati. Ad esempio si potrebbe pensare di scrivere

```
public myServletShutdown extends HttpServlet {
  // variabile privata che memorizza il numero di thread attivi
  private int serviceCounter = 0;

  // metodo per l'incremento del numero di thread attivi
  // è sincronizzato per garantire la coerenza
  protected synchronized void addRequest () {
    serviceCounter++;
  }

  // metodo per il decremento del numero di thread attivi
  // è sincronizzato per garantire la coerenza
  protected synchronized void removeRequest () {
    serviceCounter--;
  }

  // metodo per ottenere il numero di thread attivi
  // è sincronizzato per permettere di ricavare un  valore coerente
  protected synchronized int getTotRequest() {
    return serviceCounter;
  }
}
```

In questo esempio la variabile `serviceCounter` verrà incrementata ogni volta che viene effettuata una chiamata da parte del client e decrementata quando il metodo `service()` termina. In questo caso il `service()` può essere modificato nel seguente modo

```
public void service(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
  addRequest();
  try {
    super.service(req, resp);
  }
  finally {
    removeRequest();
  }
}
```

Come si può notare in questo caso, `service()` effettua una invocazione a `service()` della servlet genitore, effettuando però sia prima che dopo le modifiche alla variabile contatore.

Per effettuare invece lo shutdown indolore di una servlet è necessario che il metodo destroy() non distrugga indistintamente ogni risorsa condivisa senza prima controllare se questa sia utilizzata da qualcuno.

Un primo controllo da effettuare, necessario ma non sufficiente, è quello sulla variabile serviceCounter come nell'esempio visto poco sopra. Infatti si dovrà effettuare notifica a tutti i thread in esecuzione di un task particolarmente lungo che è arrivato il momento di concludere il lavoro.

Per ottenere questo risultato, si può far ricorso a un'altra variabile al solito accessibile e modificabile per mezzo di metodi sincronizzati. Ad esempio:

```
public mySerlvetShutdown extends HttpServlet {
  private boolean shuttingDown;
  ...
  // Metodo di modifica per l'attivazione
  // o disattivazione dello shutdown
  protected setShuttingDown(boolean flag) {
    shuttingDown = flag;
  }

  // metodo per la lettura dello stato dello shutdown
  protected boolean isShuttingDown() {
    return shuttingDown;
  }
}
```

Un esempio di utilizzo della procedura pulita per lo shutdown potrebbe essere quella mostrata qui di seguito

```
public void destroy() {

  // controlla se vi sono ancora servizi in esecuzione
  // e in tal caso invia un messaggio di terminazione
  if (numServices() > 0) {
    setShuttingDown(true);
  }

  // Attende lo stop dei vari servizi
  while(numServices() > 0) {
    try {
      Thread.sleep(interval);
    }
    catch (InterruptedException e) {
      System.out.println("Si è verificata una eccezione");
    }
```

```
    }
}
```

Questa porzione di codice, che potrebbe apparire per certi versi piuttosto oscura, assume significato se si prende in considerazione quella che è la terza raccomandazione da seguire, ovvero di implementare sempre una politica "corretta" (nel senso del termine inglese *polite*) di gestione dei vari thread.

Secondo tale regola tutti i metodi di thread che possono effettuare computazioni lunghe dovrebbero di tanto in tanto controllare lo stato di un flag globale, qui implementato con la variabile shuttingDown e, se necessario, interrompere l'esecuzione del task attivo in quel momento. Ad esempio, si può pensare di scrivere

```
public void doPost(...) {
  // esegue una computazione molto lunga
  for(i = 0; ((i < BigNumber) && !isShuttingDown()); i++) {
    try {
    // esegue una qualsiasi operazione costosa
    executionTask(i);
    }
    catch (InterruptedException e) {
      System.out.println("Si è verificata una eccezione ");
    }
  }
}
```

In questo caso l'esecuzione del lavoro a lungo termine viene effettuata direttamente all'interno del metodo doPost, piuttosto che in un thread separato. Questa soluzione, che per certi versi potrebbe apparire non troppo elegante, permette un controllo maggiore sull'esecuzione delle varie parti.

Il mantenimento dello stato

Per mantenimento dello stato si intende l'utilizzo di una qualche tecnica che permetta di mantenere informazioni fra due o più invocazioni successive di una servlet. Il protocollo HTTP infatti è di tipo *stateless* e quindi al termine della esecuzione del metodo service() la servlet perde il riferimento del client specifico e non è in grado di utilizzare informazioni prodotte precedentemente.

Si immagini ad esempio il caso in cui un client debba effettuare due o più invocazioni della stessa servlet, in modo tale che, dopo la prima invocazione, le successive debbano utilizzare i valori prodotti come risposta della prima invocazione.

Si dovrà allora predisporre un sistema che salvi i risultati intermedi o sul client, oppure sul server in un qualche modo, associando ad ogni entry nel registry una chiave che identifichi il client a cui fa riferimento.

Di seguito sono riportate le principali soluzioni adottate per ottenere questo scopo.

Modifica della invocazione

Si supponga il caso in cui il client effettui una prima invocazione della servlet inviando alcuni parametri per mezzo di un form HTML. La servlet invierà quindi in risposta al client una pagina HTML contenente un secondo form con il quale il client potrà effettuare una seconda invocazione. L'obiettivo è che questa seconda invocazione tenga conto anche della prima, ovvero che sia mantenuta memoria dei parametri della prima invocazione: un modo per fare questo potrebbe essere di modificare la pagina HTML inviata in risposta alla prima interrogazione.

Si supponga di avere un form HTML il cui codice potrebbe essere quello visto in precedenza

```
<FORM METHOD="POST" ACTION=="http://www.mokabyte.it/servlet/myservlet">
  Immettere lo user name<INPUT TYPE="text" NAME="user_id"><BR>
  <input type="submit" name= "articolo" value="ok">
</FORM>
```

Se dopo la prima invocazione la servlet vorrà memorizzare nel client per le invocazioni successive il valore "user_id=bibabc", calcolato alla prima invocazione, allora potrà modificare la pagina di risposta in modo che sia

```
<FORM METHOD="POST" ACTION="http://www.mokabyte.it/servlet/MyServlet">
  Quali articoli vuoi comprare? <INPUT TYPE="text" NAME="data"><BR>
  <input  type="submit" name="articolo" value="ok">
  <input  type="hidden" name="user_id" value="bibabc">
</FORM>
```

In questo caso il tag HTML hidden consente la creazione di un campo non visibile nella finestra del browser, ma che andrà a comporre a tutti gli effetti il set dei parametri dell'invocazione successiva.

Discorso del tutto analogo nel caso in cui la pagina di risposta non contenga un form, ma un semplice link per l'invocazione successiva della servlet: in questo caso infatti sarà necessario modificare il link presente nella pagina; ad esempio da

```
http://www.mokabyte.it/servlet/MyServlet?parametro_nuovo=valore
```

a

```
http://www.mokabyte.it/servlet/MyServlet?parametro_vecchio=valore&parametro_nuovo=valore
```

Questa tecnica, pur avendo il pregio di funzionare anche se il browser non ha attivato il supporto per i cookie (vedi oltre) non è molto utilizzata a causa della mancanza di riservatezza e sicurezza (nell'esempio specifico i dati di login sono memorizzati in chiaro nella pagina) e perché mal gestibile (è necessario implementare un qualche meccanismo di parsing e riscrittura della

pagina HTML). Una evoluzione molto comoda e potente di questo meccanismo è quella che fa uso del metodo sendRedirect(), di cui si parla poco più avanti.

Cookie

Una prima alternativa più evoluta al meccanismo appena visto è quella basata sull'utilizzo dei cookie, oggetti ben noti alla maggior parte dei navigatori internet a causa anche dell'impopolarità che hanno riscosso nei primi tempi.

Un *cookie* (letteralmente: "dolcetto", "biscotto") permette la memorizzazione di informazioni nella memoria del browser: la servlet può inviare un cookie al browser aggiungendo un campo nell'header HTTP. In modo del tutto simmetrico, leggendo tale campo è in grado di ricavare il valore dei vari cookie memorizzati in sessione.

Prendendo ad esempio il caso, peraltro piuttosto frequente, dell'implementazione di un carrello virtuale della spesa, si potrebbe avere un cookie ArticleInBasket che assume il valore 231_3, a indicare che sono stati aggiunti al carrello 3 oggetti il cui codice sia 231.

È plausibile inoltre avere per lo stesso nome di cookie valori multipli: ad esempio accanto al valore 231_3, la servlet potrebbe memorizzare anche il valore 233_4, sempre con l'obiettivo di memorizzare l'aggiunta al carrello di 4 istanze dell'oggetto il cui codice sia 233.

Secondo le specifiche del protocollo HTTP, un browser dovrebbe permettere la memorizzazione di almeno 20 cookie per host remoto, di almeno 4 kilobyte ciascuno. È questa una indicazione di massima, ed è comunque consigliabile far riferimento alle specifiche dei vari browser. In alcuni casi il server, a seconda delle impostazioni utente e delle sue stesse caratteristiche, può ottenere solamente i cookie da lui memorizzati nel client: in questo caso tutte le servlet in esecuzione all'interno dello stesso server possono di fatto condividere i vari cookie memorizzati.

In Java si possono creare e gestire i cookie tramite la classe javax.serv-let.http.Cookie: ad esempio per la creazione è sufficiente invocare il costruttore

```
Cookie C = new Cookie("uid","bibabc")
```

In tal caso si memorizza un cookie nel browser in modo da tener memoria della coppia variabile, valore "uid=bibabc". Eventualmente è possibile modificare successivamente tale valore per mezzo del metodo setValue().

Il nome di un cookie deve essere un token dello standard HTTP/1.1: in questo caso per token si intende una stringa che non contenga nessuno dei caratteri speciali menzionati nella RFC 2068, o quelli che iniziano per $ che sono invece riservati dalla RFC 2109.

Il valore di un cookie può essere una stringa qualsiasi, anche se il valore null non garantisce lo stesso risultato su tutti i browser. Se si sta inviando un cookie conforme con la specifica originale di Netscape, allora non si possono utilizzare i caratteri

```
[ ] ( ) = , " / ? @ : ;
```

Si possono inoltre settare alcuni parametri e informazioni del cookie stesso, come il periodo massimo di vita, metodo setMaxAge(), periodo dopo il quale il cookie viene eliminato dalla cache

del browser; è possibile anche inserire un commento tramite il metodo setComment() che verrà presentato all'utente nel caso questo abbia abilitato il controllo: questa non è una feauture standard e quindi non è trattata nello stesso modo da tutti i browser.

Una volta creato, per aggiungere il cookie al browser si può utilizzare il metodo

```
HttpServletResponse.addCookie(C)
```

Ad esempio per creare un cookie e inviarlo al browser

```
public void doGet (HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException {
  Cookie getItem = new Cookie("Buy", ItemId);
  getBook.setComment("Oggetto acquistato dall'utente ");
  response.addCookie(getBook);
}
```

Nel caso in cui la servlet restituisca una risposta all'utente per mezzo dell'oggetto Writer, si deve creare il cookie prima di accedere al Writer: infatti è noto che le operazioni relative all'header, dove di fatto risiedono i cookie, devono essere effettuate prima dell'invio dei dati veri e propri. Ad esempio si dovrà scrivere

```
public void doGet (HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException {

  // crea un cookie
  Cookie getItem = new Cookie("Buy", ItemId);

  // imposta il content-type header prima di acccedere al Writer
  res.setContentType("text/html");

  // ricava il Writer e scrive i dati della risposta
  PrintWriter out = response.getWriter();
  out.println("<html><head><title> Book Catalog </title></head>");
}
```

L'operazione opposta, ovvero ricavare i vari cookies memorizzati nel browser, può essere effettuata grazie ai metodi

```
public Cookie getCookies(String CookieName)
public Cookie[] getCookies()
```

Il primo restituisce un cookie identificato per nome, mentre il secondo tutti quelli memorizzati ed è presente nella classe javax.servlet.http.HttpServletRequest.

```
Cookie[] cookies = req.getCookies();
```

```
for(i=0; i < cookies.length; i++) {
  Cookie thisCookie = cookie[i];
  if (thisCookie.getName().equals("compra")) {
    System.out.println("trovato il cookie " + thisCookie.getValue());
    // rimuove il cookie impostando l'età massima a 0
    thisCookie.setMaxAge(0);
  }
}
```

In questo caso dopo aver ricavato tutti i cookie disponibili, si stampa il valore in essi memorizzato per mezzo del metodo getValue().

È possibile invalidare un cookie semplicemente impostando la sua età massima a zero.

Si deve dire che, per quanto flessibile e semplice, la soluzione dei cookie spesso non viene scelta a causa del basso livello di sicurezza offerto. Infatti il cookie, e quindi anche le informazioni in esso contenute, sono memorizzate sul client: questa situazione può consentire a malintenzionati di ricavare tali informazioni durante il tragitto dal client al server (ad esempio durante una invocazione).

Inoltre se in una intranet è abbastanza plausibile aspettarsi che tutti i browser supportino i cookie o comunque siano configurati per usarli, una volta in internet non si può più fare affidamento su tale assunzione, per cui vi possono essere casi in cui tale tecnica può non funzionare.

Le sessioni

Il concetto di sessione, introdotto da Java proprio con le servlet, permette di memorizzare le informazioni su client in modo piuttosto simile a quanto avviene ad esempio per i cookie: la differenza fondamentale è che le informazioni importanti sono memorizzate all'interno di una sessione salvata sul server, invece che sul client, al quale viene affidato solamente un identificativo di sessione.

Dato che non si inviano dati al client come invece avviene con i cookie, si ha una drastica riduzione del traffico in rete con benefici sia per le prestazioni che per la sicurezza.

Per associare il client con la sessione salvata, il server deve "marcare" il browser dell'utente, inviando un piccolo e leggero cookie sul browser con il numero di identificazione della sessione, il SessionId.

Le interfacce che permettono la gestione delle sessioni sono la HttpSession, la HttpSessionBindingListener e la HttpSessionContext.

Per creare un oggetto di tipo Session, si può utilizzare il metodo getSession() della classe HttpServletRequest, come ad esempio:

```
public class SessionServlet extends HttpServlet {

public void doGet (HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {
  // ricava o crea la sessione del client
  HttpSession session = req.getSession(true);
  out = response.getWriter();
}
```

Il valore booleano passato alla getSession() serve per specificare la modalità con cui ottenere la sessione: nel caso di un true, la servlet tenterà prima di ricavare la sessione se questa è già presente, altrimenti ne creerà una nuova. Il valore false invece permette di ottenere sessioni solo se queste sono già presenti.

Con la versione 2.1 della API è stato introdotto per comodità il metodo

```
request.getSession();
```

senza parametri, la cui invocazione ha lo stesso effetto di quella con parametro true.

Nel caso in cui la sessione venga creata ex novo, nel browser verrà memorizzato un cookie contenente l'identificativo univoco della sessione stessa. Per una sessione appena creata, il metodo isNew() restituirà true: in questo caso la sessione non contiene nessuna informazione.

Da notare che, sempre con la versione 2.1 della API l'interfaccia HttpSessionContext (che fornisce una interfaccia verso il contesto di sessione) è stata deprecata, così come il metodo HttpSessionContext.getSessionContext() che adesso restituisce una sessione nulla che non contiene informazioni.

Il motivo di questa scelta è legato alla necessità di garantire un livello maggiore di sicurezza, dato che un suo utilizzo permette a una servlet di accedere ed eventualmente modificare le informazioni memorizzate in una sessione creata a gestita da un'altra servlet.

Altra innovazione portata con la API 2.1 è la possibilità di impostare, oltre che con i tool forniti dal server, la durata massima della validità di una sessione per mezzo del metodo

```
public void setMaxInactiveInterval(int interval)
```

fornito direttamente dalla HttpSession, insieme all'analogo

```
public int getMaxInactiveInterval()
```

Come avviene con i cookie, al fine di mantenere una sessione in modo corretto, si dovrà fare attenzione a invocare getSession() prima di tutte le operazioni di scrittura verso il client, dato che questa cosa blocca tutte la possibilità di agire sull'header e quindi di creare una sessione.

Una volta ottenuto l'oggetto Session, è possibile usarlo per memorizzare qualsiasi tipo di informazione relativo alla sessione logica che essa rappresenta, per mezzo degli opportuni metodi della HttpSession.

Ad esempio, per contare il numero di volte che una servlet viene invocata, sarà sufficiente scrivere

```
// si prova a ricavare dalla sessione
// il valore di sessiontest.counter
Integer ival = (Integer) session.getValue("sessiontest.counter");

// se la sessione non esiste il valore restituito sarà null
if (ival==null)
```

```
    // ival non è definito
    ival = new Integer(1);
  else {
    // il valore di ival è definito
    // si incrementa di 1
    ival = new Integer(ival.intValue() + 1);
  }

  // si memorizza nella sessione il nuovo valore
  session.putValue("sessiontest.counter", ival);
```

Dunque Session è una specie di dizionario di valori associati a un'unica sessione HTTP. Questa è un'altra importante differenza, con indubbio vantaggio sulla semplicità, rispetto ai cookie, dove per ogni valore che si vuole memorizzare si deve installare un cookie apposito.

Cancellazione di una sessione

Una sessione utente può essere invalidata manualmente mediante il metodo invalidate(), oppure se supera il tempo massimo di inattività.

Invalidare una sessione significa eliminare l'oggetto di tipo HttpSession e quindi tutti i dati in esso memorizzati. I motivi per cui una sessione deve essere eliminata possono essere molteplici: tipicamente al logout dell'utente dalla applicazione si procede sempre alla rimozione di tutti i dati ad esso associati, ovvero alla rimozione della sessione.

Session tracking e disattivazione dei cookie

La gestione delle sessioni, pur essendo sicuramente uno strumento più potente del semplice cookie, fa sempre uso di tali oggetti: infatti, benché le informazioni siano memorizzate sul server, il SessionID viene memorizzato nel client per mezzo di un cookie.

Volendo permettere il corretto funzionamento di una applicazione basata su servlet, indipendentemente dal fatto che il browser sia abilitato o meno al supporto per i cookie, si può ricorrere a una tecnica detta "URL-rewriting": in questo caso la servlet dovrà modificare la pagina HTML da inviare in risposta al client inserendo l'id della sessione come parametro per le invocazioni successive sulla servlet stessa.

Ad esempio se il link alla servlet contenuta nella pagina fosse qualcosa del tipo

```
http://nome_host/servlet/nomeservlet
```

lo si dovrà trasformare in

```
http://nome_host/servlet/nomeservlet$SessionID=234ADSADASZZZXXXSWEDWE$
```

Questa trasformazione può essere effettuata in modo del tutto automatico grazie ai metodi encodeURL() ed encodeRedirectURL() entrambi messi a disposizione dalla HttpServletResponse.

Più precisamente nel caso in cui si debba semplicemente codificare un URL con il session id, si potrà utilizzare il primo, mentre nel caso di una redirect verso un'altra pagina si potrà fare uso del secondo.

L'utilizzo di tali metodi è piuttosto semplice: infatti, supponendo che la prima servlet effettui una stampa riga per riga della pagina da inviare al client, il metodo service() potrebbe diventare

```
public void doGet (HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException {

  // Ricava la sessione utente, il Writer, ecc.
  ...

  // inizia la scrittura della pagina
  out.println("<html>" + ...);
  . . .
  // scrive il codice HTML relativo a un link immettendo SessionID
  out.print("<a href=");
  out.print(response.encodeURL("/servlet/servlet2?p1="+p1+"));
  out.println("> </a>");
  // effettua la stampa di altro codice HTML
  . . .
  // scrive il codice HTML relativo a un secondo link
  // immettendo il SessionID
  out.print("<a href=");
  out.print(response.encodeURL("/servlet/servlet2?p2="+p2+"));
  out.println("> </a>");
  . . .
  // conclude la scrittura della pagina
  out.println("</html>");

}
```

Da notare che la servlet deve effettuare la modifica di tutti i link che verranno mostrati nella pagina HTML inviata al client.

Fatto molto interessante è che la ricodifica dell'URL avviene in maniera selettiva in funzione del fatto che il client abbia o meno abilitato il supporto per i cookie.

Da un punto di vista operativo, nel momento in cui l'utente effettua un click sul link modificato, il metodo getSession() della servlet ricava automaticamente la sessione dall'URL, in modo analogo al caso in cui la sessione sia stata memorizzata nella cache del browser sotto forma di cookie.

La tecnica dell'URL-rewriting funziona anche con pagine JSP, e anzi viene utilizzata molto spesso all'interno di tali oggetti: in questo caso, in una pagina JSP tutti i link dovranno essere ridefiniti in modo che gli URL associati siano modificati automaticamente; ad esempio nella applicazione di gestione della Community di MokaByte la pagina principale di menu contiene il seguente link

```
<tr>
  <td>
    <a href='<%=response.encodeRedirectURL("addnew.show")%>'>Registrazione</a>
    <br>Creazione di un nuovo profilo utente della Community di MokaByte
  </td>
</tr>
```

che permette di accedere alla procedura di registrazione di un nuovo utente.

Web Application

Disponibili con la versione 2.2 della Servlet API le web application rappresentano forse una delle novità più interessanti introdotte da Sun, anche se non sono esclusivamente legate alle servlet.

Una web application, come definita nella Servlet Specification, è una collezione di servlet, Java Server Pages, classi Java di utilità, documenti statici come ad esempio pagine HTML, raggruppati all'interno di un archivio JAR in modo da semplificare al massimo le operazioni di installazione e configurazione. Se in passato per effettuare il deploy e la configurazione di una applicazione basata su servlet era necessario copiare una serie di file ed editarne altri, adesso, con le web application, è sufficiente copiare un unico file JAR in una directory opportuna del server: all'interno di tale file (che ha estensione .war) sono contenute tutte le risorse necessarie alla applicazione (.class, .html, immagini) ma anche tutte le informazioni per la configurazione dell'applicazione stessa. Chiunque abbia provato a utilizzare la vecchia modalità di installazione e configurazione si renderà conto di quanto questo possa essere comodo e vantaggioso.

Si limiterà comunque in questo capitolo la trattazione delle web application agli aspetti legati alla programmazione delle servlet.

WAR files

Il file JAR che contiene una web application in questo caso prende il nome di Web Application Archive (WAR file) e ha estensione .war.

Una struttura tipica di un file di questo tipo potrebbe essere ad esempio

```
index.htm
index.jsp
images/title.gif
images/mokalogo.gif
WEB-INF/web.xml
WEB-INF/lib/jspbean.jar
WEB-INF/classes/MyServlet.class
WEB-INF/classes/it/mokabyte/MokaServlet.class
```

Al momento dell'installazione, il file .war deve essere posizionato in una directory mappata poi dal server HTTP con un URI particolare. Tutte le richieste inoltrate a tale URI saranno poi gestite dall'applicazione contenuta in tale file WAR.

La directory WEB-INF

La directory WEB-INF ha un compito piuttosto particolare all'interno di una web application: i file qui contenuti infatti non sono accessibili direttamente dai client, ma sono utilizzati dal server per configurare l'applicazione stessa. Il suo funzionamento è quindi molto simile a quello della directory META-INF di un normale file JAR.

La sottodirectory WEB-INF\classes contiene tutti i file compilati delle varie servlet e delle classi di supporto; la WEB-INF\lib invece contiene altre classi che sono però memorizzate in archivi JAR. Tutte le classi contenute in queste due directory sono caricate automaticamente dal server al momento del load della applicazione stessa.

Le servlet presenti nella dir WEB-INF possono essere invocate tramite un URL che, nel caso dell'esempio appena visto, potrebbe essere

```
/<application_name>/servlet/MyServlet
/<application_name>/servlet/it.mokabyte.MokaServlet
```

Il file web.xml contenuto nella dir WEB-INF è noto come *deployment descriptor file*, e contiene tutte le informazioni relative all'applicazione in cui risiede. Si tratta di un file XML il cui DTD è fissato da Sun con una specifica che indica oltre 50 tag con i quali poter specificare una qualsiasi delle seguenti informazioni:

- Le icone per la rappresentazione grafica dell'applicazione.
- La descrizione dell'applicazione.
- Un flag che indica se la applicazione è *distributed*, ovvero se tale applicazione può essere condivisa fra diversi server remoti. I motivi per cui possa essere utile realizzare un'applicazione distribuita sono molti, il più importante dei quali è forse legato a una maggiore flessibilità e alla possibilità di implementare tecniche di balancing dinamico e distribuzione del carico computazionale. Le regole per progettare e scrivere un'applicazione distribuita sono molte ed esulano dallo scopo di questo capitolo, per cui si rimanda per maggiori approfondimenti alla bibliografia ufficiale.
- I parametri di inizializzazione dell'applicazione nel complesso, o delle singole servlet.
- La registrazione del nome o dei nomi di una servlet in esecuzione: possibilità questa che semplifica e soprattutto standardizza la fase di deploy di una servlet.
- L'ordine di caricamento delle servlet.

- Le regole di mapping fra le varie servlet e i relativi URL.

- Il timeout delle sessioni, grazie ai nuovi metodi di gestione delle sessioni introdotti con la API 2.2.

- Il welcome file list, ovvero la sequenza dei file da utilizzare per la risposta da inviare al client (p.e.: index.htm, index.html o welcome.htm).

- Le regole per la gestione degli errori, tramite le quali specificare fra l'altro anche quali pagine HTML (statiche o dinamiche) debbano essere visualizzate in corrispondenza del verificarsi di un errore del server (p.e. una pagina non trovata) o di una eccezione prodotta dal servlet engine.

- I riferimenti aggiuntivi alle tabelle di lookup utilizzate durante i riferimenti con JNDI a risorse remote.

- Le regole di policy, tramite le quali specificare vincoli aggiuntivi di sicurezza.

Da un punto di vista pratico, la struttura del file XML non è importante di per sé, ma piuttosto è importante il fatto che la sua presenza permette finalmente la possibilità di automatizzare e standardizzare il processo di installazione e deploy della applicazione così come di un gruppo di servlet.

Grazie al meccanismo di installazione e deploy messo in atto con l'utilizzo dei file .war, si parla sempre più spesso di Pluggables Web Components, ovvero di componenti con funzionalità specifiche installabili in un application server in modo molto semplice e veloce.

Si pensi ad esempio alla possibilità di attivare un motore di ricerca semplicemente copiando un file in una directory opportuna, indipendentemente dal sistema operativo e dal server. Parallelamente, una azienda che desideri attivare servizi di hosting a pagamento potrà utilizzare il sistema delle web application e dei file .war per semplificare la gestione degli utenti e dei vari domini governati da servlet.

La programmazione di una web application

Dal punto di vista della programmazione, una web application può essere vista come un ServletContext, di cui si è parlato in precedenza; tutte le servlet all'interno della medesima web application condividono lo stesso ServletContext.

I parametri di inizializzazione specificati tramite il file descrittore di deploy possono essere ricavati per mezzo dei metodi della interfaccia ServletContext

```
public String getInitParameter(String name)
public Enumeration getInitParameterNames()
```

che sono gli equivalenti dei metodi omonimi della classe GenericServlet.

Una servlet può ricavare il prefisso dell'URL rispetto al quale è stata invocata tramite il nuovo metodo getContextPath() fornito dalla ServletRequest.

In questo caso viene restituita una stringa che rappresenta il prefisso dell'URL relativo al context in esecuzione: ad esempio per una richiesta a una servlet in esecuzione in

```
/moka/articles/servlet/FinderServlet
```

il getContextPath() restituirà la seguente stringa

```
/moka/articles
```

Ogni servlet conosce già il suo contesto, per cui non è necessario includerlo tutte le volte che si vuole accedere alle risorse contenute nel contesto stesso se non si desidera dover effettuare una ricompilazione tutte le volte che si esegue una qualche modifica all'applicazione.

Gestione della sicurezza

Spesso accade di dover realizzare web application basate su servlet, su pagine JSP (più comunemente su entrambe) in cui l'accesso dell'utente a determinate risorse debba essere subordinato ad una serie di vincoli in modo da garantire un opportuno livello di sicurezza.

Più precisamente può essere necessario poter concedere l'accesso solo a determinate persone (*authentication*) e in base al profilo utente fornire l'accesso solo a specifiche risorse (*authorization*). Inoltre spesso accade di dover fornire la certezza che solo le parti coinvolte nella comunicazione, l'utente e il server, possano accedere in modo riservato alle informazioni scambiate fra i due (*confidentiality*). Strettamente legato a quest'ultimo aspetto vi è quello della integrità delle informazioni: nessuno deve poterne alterare il contenuto durante il tragitto fra utente e server, i quali devono poterne verificare l'integrità (*integrity*).

Le considerazioni che verranno qui esposte non sono prerogativa esclusiva del mondo delle servlet, ma sono in genere valide per tutti quei casi in cui una risorsa statica (un file HTML o di testo) o dinamica (servlet, pagine JSP) sia accessibile dall'esterno e se ne debbano configurare gli aspetti legati alla sicurezza appena accennati.

Per questo, quanto qui affrontato verrà visto in ottica web application, spazio in cui di fatto una servlet vive. Alcune tecniche di programmazione in realtà non hanno a che fare con il processo di configurazione delle servlet o di Java ma più spesso sono procedure di personalizzazione del server web e della web application.

Nel prosieguo del paragrafo verranno presentate le tecniche standard per proteggere le risorse e per certificare l'identità di server e client cercando di semplificare al massimo il discorso limitando la trattazione al caso di semplici web application e semplici servlet. L'argomento della gestione della sicurezza è molto vasto e in alcuni punti esula dallo scopo di questo capitolo. Per chi fosse interessato a maggiori approfondimenti, si rimanda alla bibliografia.

Autenticazione e protezione di risorse

Per semplificare il discorso si ipotizzerà di dover proteggere una servlet dall'accesso pubblico e che tale servlet viva all'interno di una web application denominata secret.

L'obiettivo finale è quello di fornire accesso alla servlet solo se l'utente fornisce le corrette credenziali: si deve quindi utilizzare un meccanismo che permetta l'autenticazione e la autorizzazione all'accesso a determinate risorse.

Prendendo spunto dal meccanismo degli utenti e gruppi tipico di Unix, Java utilizza un sistema di autenticazione e autorizzazione basato sui ruoli: solamente dopo aver accertato che un utente appartiene a un determinato ruolo, si fornisce l'accesso solo alle risorse (statiche o dinamiche) autorizzate per quel determinato ruolo.

Il processo di autenticazione può essere implementato in vari modi: nel caso più semplice questo comporta l'inserimento di userid e password, mentre volendo adottare un modello con uno standard di sicurezza più alto si potrà implementare un meccanismo di riconoscimento reciproco basato su chiavi e certificati digitali.

Una volta accertato chi sia l'utente, il servlet container agisce utilizzando il concetto di security constraint, il quale associa il ruolo a un insieme di risorse.

Il servlet container in genere utilizza vari meccanismi per definire il set di utenti e associare i ruoli: la forma più semplice, utilizzata ad esempio anche da Tomcat, è quella che prevede l'uso di un file in formato XML in cui sono memorizzate userid, password e ruolo. Ad esempio, in Tomcat al momento della installazione sono definiti alcuni utenti all'interno del file tomcat-users.xml, presente nella directory conf della installazione; tale file potrebbe avere la seguente struttura:

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
<role rolename="tomcat"/>
<role rolename="role1"/>
<role rolename="manager"/>
<user username="tomcat" password="tomcat" roles="tomcat,manager"/>
<user username="both" password="tomcat" roles="tomcat,role1"/>
<user username="role1" password="tomcat" roles="role1"/>
</tomcat-users>
```

In questo caso manager è per definizione il ruolo corrispondente all'amministratore del server ed è quello che si usa fra le altre cose per accedere alla console di amministrazione del server stesso. Il file tomcat-users.xml permette di definire una terna composta da (username,password,ruolo) tramite il tag <username>.

Si noti che, essendo il file utilizzato come configurazione del server, non è possibile in questo caso definire meccanismi di accesso personalizzati ai dati per ogni singola applicazione: nel caso in cui questo fosse necessario è necessario ricorrere a tecniche più complesse descritte in seguito.

Una volta definito un utente o un gruppo di utenti, all'interno della applicazione si dovranno definire le seguenti impostazioni: modalità di login, insieme di risorse della applicazione da pro-

teggere associandole a determinati ruoli, sistema di trasporto dei dati. Tali informazioni sono inserite tramite appositi tag all'interno del file web.xml.

I modelli di autenticazioni di una risorsa web sono i seguenti:

- BASIC e FORM: in questo caso l'autenticazione avviene tramite l'utilizzo di ruoli (*roles*) associati a determinati utenti e il meccanismo di riconoscimento avviene utilizzando le funzionalità offerte direttamente dal container.

- Custom: in questo caso il processo di riconoscimento viene implementato in modo manuale dal programmatore che può far uso delle API offerte da Java in tema di sicurezza o ricorrere a tecniche personalizzate.

- DIGEST: basato su certificati digitali

Autenticazione BASIC e FORM

Questi due casi sono molto simili fra loro e utilizzano una particolare configurazione del server HTTP in modo che esso richieda userid e password al momento in cui l'utente tenta di accedere ad una risorsa protetta (non solo servlet, ma anche un file HTML o una immagine GIF).

Questa modalità di protezione molto spesso viene messa in atto utilizzando le funzionalità del server HTTP più che il servlet container.

Sia che si usi l'uno o l'altro, il meccanismo di autenticazione è simile: il server controlla prima se l'utente ha già effettuato un processo di autenticazione (login) e, in tal caso, se il ruolo associato corrisponde a quello necessario per poter accedere alla risorsa. Nel caso in cui si effettui l'autenticazione tramite un server HTTP, si deve agire direttamente sulla configurazione del server in modo dipendente dal particolare server utilizzato: in Apache, ad esempio, tramite il file httpd.conf. Se invece si adotta l'autenticazione attuata dal servlet container, la procedura di configurazione è standard e definita dalla specifica: la definizione di una risorsa protetta e l'associazione al ruolo avviene all'interno del file web.xml che potrebbe essere così definito:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <servlet>
    <servlet-name>secretservlet</servlet-name>
    <servlet-class>com.mokabyte.mokabook.servlets.security.basic.SecretServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>secretservlet</servlet-name>
    <url-pattern>/secretservlet</url-pattern>
  </servlet-mapping>
. . .
  <security-constraint>
    <display-name>basic</display-name>
```

```
    <web-resource-collection>
      <web-resource-name>Secured Web Collection</web-resource-name>
      <url-pattern>/</url-pattern>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
      <role-name>manager</role-name>
    </auth-constraint>
    <user-data-constraint>
      <transport-guarantee>NONE</transport-guarantee>
    </user-data-constraint>
  </security-constraint>
  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>default</realm-name>
  </login-config>
  <security-role>
    <description>una autorizzazione per il manager</description>
    <role-name>manager</role-name>
  </security-role>
. . .
</web-app>
```

In questo caso nel deployment descriptor dell'applicazione viene definita la servlet com.mokabyte.mokabook.servlets.security.basic.SecretServlet alla quale è associato il nome secret. L'URL mapping con il quale è possibile invocare la servlet è /secretservlet.

Il tag <security-constraint> inizia la definizione di tutti gli aspetti legati alla sicurezza. Il tag <web-resource-collection> definisce il set di risorse che si vogliono proteggere: in questo caso tramite il tag <url-pattern>

```
<url-pattern>/</url-pattern>
```

si proteggono tutti gli URL associati alle applicazioni; possono essere presenti un numero arbitrario di coppie <url-pattern>. Tramite i tag

```
<http-method>GET</http-method>
<http-method>POST</http-method>
```

entrambi i metodi di invocazione GET e POST saranno soggetti alle restrizioni imposte.

Il pezzo di script XML

```
<auth-constraint>
  <role-name>manager</role-name>
</auth-constraint>
```

definisce invece un ruolo da associare al constraint: in questo caso manager potrà accedere alle risorse protette e definite in <web-resource-collection>.

Infine, il tag <login-config> permette di specificare il tipo di autenticazione richiesta: in questo caso la forma BASIC, gestita completamente dal servlet container.

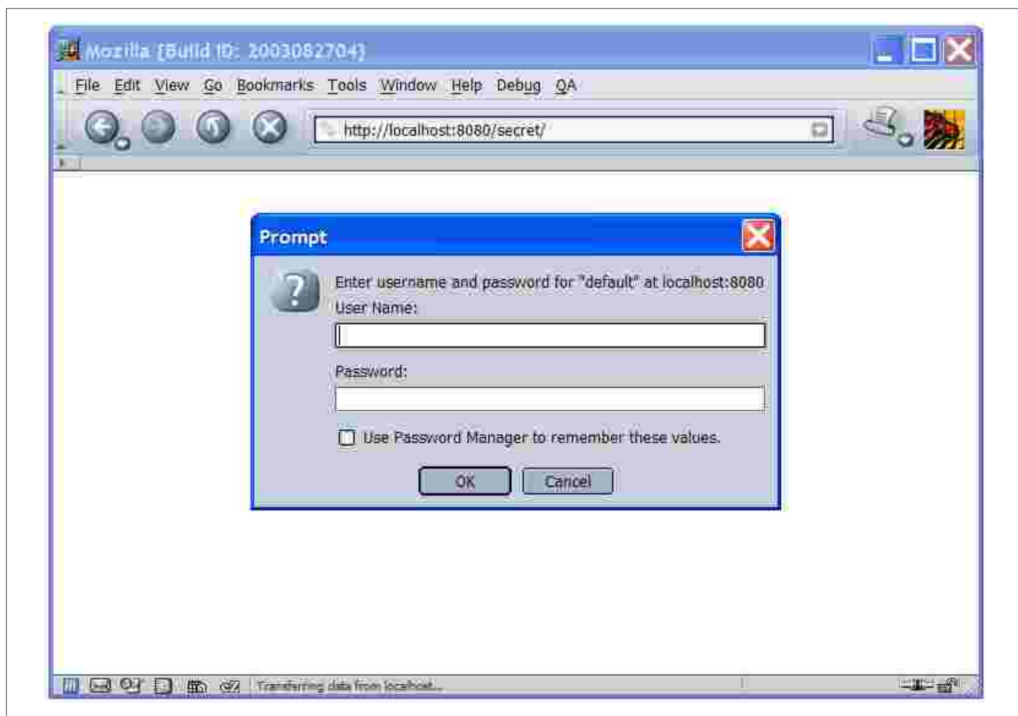
```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>default</realm-name>
</login-config>
```

Il <realm-name> serve per specificare quale dominio applicativo debba essere associato al security constraint, cosa utile nel caso in cui l'applicazione serva per più domini.

Infine segue l'elenco dei ruoli ammessi per questa applicazione.

```
<security-role>
  <description>una autorizzazione per il manager</description>
```

Figura 5.7 – *Al momento dell'autenticazione BASIC, il browser mostra una finestra di dialogo in cui immettere userid e password.*

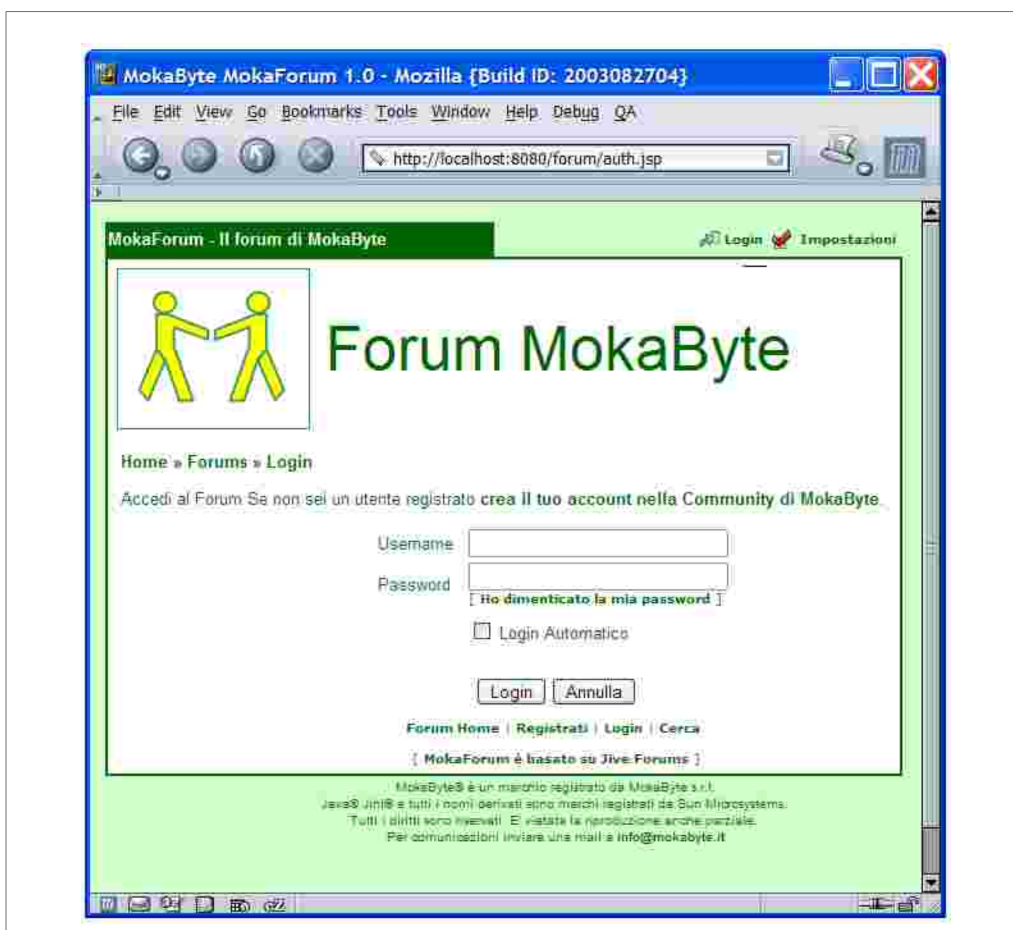


```
  <role-name>manager</role-name>
</security-role>
```

Nel caso di autenticazione BASIC, all'utente verrà presentata la classica finestra del browser in cui si chiede di immettere i dati di autenticazione (userid e password).

Questo meccanismo di autenticazione non è quasi più utilizzato, dato che è poco user friendly e certamente non più "alla moda". Ormai tutte le applicazioni web mostrano una pagina HTML

Figura 5.8 – *Con l'autenticazione di tipo FORM è possibile visualizzare all'utente una pagina HTML contenente il form di registrazione. In questo caso l'utente può ricevere un numero maggiore di informazioni e si troverà più a suo agio dato che la pagina di login ha lo stesso look and feel del resto dell'applicazione web.*



personalizzata in cui inserire i dati di accesso. Questo permette di mostrare all'utente una pagina coerente con il resto dell'applicazione, di mostrare anche maggiori dettagli informativi di aiuto e pure una pagina di errore personalizzata nel caso in cui l'utente immetta dati non validi.

In fig. 5.8 è riportato il form HTML di autenticazione richiesto per poter entrare all'interno del forum di MokaByte: in questo caso l'utente riceve una pagina gradevole ed esplicativa delle informazioni da eseguire; nel caso in cui l'utente non ricordi la password potrà ciccare su un link e andare nella pagina della community di MokaByte da cui farsi inviare per e-mail i suoi dati personali.

Per poter attivare il sistema di autenticazione basato su form HTML è necessario modificare il file `web.xml` introducendo al posto di `BASIC` il valore `FORM`

```
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>default</realm-name>
  <form-login-config>
    <form-login-page>/login.htm</form-login-page>
    <form-error-page>/error.htm</form-error-page>
  </form-login-config>
</login-config>
```

Si noti come in questo caso si deve specificare il nome del file HTML da utilizzare come form di login e anche la pagina di errore.

Anche nel caso di utilizzo di un form HTML, il processo di autenticazione è a completo carico del servlet container: il form infatti invoca un URL particolare (`j_security_check`) provvedendo a passare parametri particolari (`j_username` e `j_password`); sinteticamente il codice HTML necessario per definire il form è il seguente:

```
<form  action='j_security_check' name="loginForm" method="POST">
  <input type="text" name="j_username" size="16" maxlength="16" id="username"/>
  <input type="password" name="j_password" size="16" maxlength="16" id="password"/>
  <input type="submit" value="Entra" name="submit">
</form>
```

Ovviamente tutto intorno si possono mettere tutti gli elementi grafici necessari per abbellire la pagina e personalizzarla a piacimento.

Il sistema appena visto offre certamente un livello molto basso di sicurezza e non permette molte personalizzazioni: utilizzare un file di testo (`tomcat-users.xml`), oltre ad essere poco flessibile, non offre grandi garanzie di sicurezza se qualcuno riesce ad accedere a tale file di testo.

Un piccolo miglioramento si può ottenere utilizzando database relazionali, alberi LDAP o altro ancora verso i quali l'applicazione web potrà effettuare la ricerca di userid e password in vario modo. In [TOM] ad esempio viene mostrato come utilizzare un database relazionale in cui memorizzare tali informazioni e come customizzare l'applicazione per accedere a tali dati.

Recupero delle informazioni client

Ad ogni invocazione la servlet può accedere ad una serie di invocazioni relative al client all'interno dell'header HTTP.

Queste informazioni possono essere ricavate tramite alcuni metodi messi a disposizione dalla API: i metodi getRemoteUser() e getAuthType() restituiscono il nome dell'utente e il tipo di autenticazione utilizzata.

A partire dalla servlet API 2.2 è stato inserito il metodo getUserPrincipal(), che restituisce un oggetto implementazione dell'interfaccia javax.security.Principal:

```
public javax.security.Principal HTTPServletRequest.getUserPrincipal()
```

Il concetto di Principal è stato introdotto per uniformare i meccanismi di sicurezza in J2EE (è utilizzato ad esempio anche in EJB per definire le policy di accesso ai metodi di session ed entity bean): può essere un utente, un gruppo, una azienda o semplicemente un identificativo di un soggetto che accede alla risorsa.

Principal include il metodo getName() che restituisce il nome del Principal. Da notare che getUserPrincipal() deve essere preferito per mantenere omogeneità in J2EE, anche se la servlet API include getRemoteUser() per compatibilità con lo standard universalmente adottato in CGI. Infine il metodo

```
public boolean isUserInRole(String Role)
```

permette di verificare se l'utente loggato appartiene o meno a un determinato ruolo, consentendo di realizzare politiche di accesso variabili in funzione del ruolo utente.

Autenticazione di tipo custom

Molto spesso accade che per l'autenticazione non sia sufficiente affidarsi al meccanismo offerto dall'application server (modelli BASIC o FORM): se userid e password non sono le uniche informazioni da immettere o se si vuole operare in modo più personalizzato nella procedura di login, si deve per forza ricorrere a sistemi scritti appositamente per rispondere alle proprie esigenze.

In questo caso tutte le richieste fatte dagli utenti sono intercettate da questa servlet: se l'utente è loggato, la servlet inoltrerà la richiesta alla risorsa invocata dal client, altrimenti rimanderà alla pagina contenente il form di login. Tale form poi invocherà un'altra servlet che provvederà a controllare se i dati sono corretti e procedere in tal caso a creare una sessione di dati dell'utente.

Una variante più elegante di quella basata sulla "servlet intercettatutto" prevede l'utilizzo di un filtro: questi oggetti inseriti nella specifica con la API 2.3 (vedi oltre) non sono altro che particolari servlet le quali hanno il solo compito di intercettare tutte le richieste e risposte per gli URL sui quali sono mappate e di inoltrare poi la richiesta altrove.

Questa soluzione, oltre a sfruttare in modo migliore le potenzialità offerte dalla API, ha il grosso vantaggio di non impattare sui mapping definiti per le servlet o per le altre risorse dell'applicazione. Ad esempio, nelle moderne web application basate sul modello Model-View-Controller

(come Struts [STRU] o MokaPackages [MBPAK]), spesso tutte le invocazioni corrispondenti alla esecuzione della business logic sono mappate con URL del tipo `*.run` o `*.do`, mentre le invocazioni di pagine HTML dinamiche (in MVC dette "viste") sono in genere mappate con URL del tipo `*.show` o `*.view`.

A questi URL sono poi associate determinate servlet che inoltrano le richieste verso opportune risorse: nel caso di `*.run`, ad esempio, verso classi action che eseguono la business logic, mentre, per le `*.show`, verso pagine JSP.

Il motivo per il quale si adottano queste tecniche, che a una prima analisi possono sembrare prive di senso, è alla base della potenza del pattern MVC: dato che una sua trattazione esula dagli scopi di questo capitolo si rimanda a [MBMVC].

Appare evidente che mappare un'altra servlet per entrambi gli URL-pattern in modo da effettuare i controlli sui login risulta essere piuttosto scomodo e anche rischioso: si può avere la certezza che il servlet container invochi sempre prima la `LoginServlet` prima di ogni altra? E poi, sovrapponendo due URL mapping simili (`/` vince su `*.run`?) si può avere la certezza che entrambi verranno presi in considerazioni e che le rispettive servlet verranno eseguite? Per esperienza, queste situazioni portano a impredicibilità del comportamento della applicazione, sia all'iterno dello stesso application server che fra prodotti differenti.

I filtri invece, essendo un particolare tipo di servlet che per specifica sono eseguite prima di ogni altro (o dopo), non introducono variabili indeterminabili e non impattano con il set di mapping specificato per le servlet, avendone uno tutto loro.

Di seguito è riportato un esempio che mostra come potrebbe essere implementato questo meccanismo customizzato.



Fra gli esempi allegati al libro non è stato incluso un esempio che mostra come realizzare un login personalizzato. Il lettore potrà facilmente realizzare una applicazione completa partendo dalle porzioni di codice qui riportato. Una implementazione completa che supporti l'autenticazione Custom, unitamente alla gestione dei ruoli utente, è presente all'interno di MokaPackages, il framework di MokaByte in MVC; per maggiori informazioni si veda [MPACK].

La servlet che effettua il controllo potrebbe essere così implementata:

```
public void service(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException {
  . . .
  // Ricava un oggetto LoggedUser dalla sessione
  LoggedUser loggedUser = (LoggedUser) req.getSession().getAttribute("loggedUser");
  if (loggedUser != null) {
    String role = loggedUser.getUserRole();
    // page = ricava la pagina dalla request invocata dal client oppure in altro modo
    res.sendRedirect(res.encodeRedirectURL(page));
  }
```

```
    else{
      // lancia una eccezione proprietaria
      throw new UserNotLoggedException("Operazione non consentita, utente non loggato");
    }
  }
```

In questo caso il servlet mappato per rispondere a tutte le richieste dei client controlla se in sessione è presente un oggetto di tipo LoggedUser, che deve essere inserito in sessione al momento del login. Ad esempio, se fosse una servlet a effettuare il logi, si potrebbe scrivere

```
public void service(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException {
  // ricava i dati utente dalla sessione
  String UserId = httpServletRequest.getParameter("userId");
  String UserPassword = httpServletRequest.getParameter("passwd");

  // con una qualche logica di collegamento verso un database remoto effettua
  // il login passando userid e password. Viene restituito un oggetto Role associato all'utente
  // utile se si vuole implementare una logica di controllo basata sui ruoli
  Role role = login(UserId, UserPassword);

  if (role != null){
    HttpSession session = httpServletRequest.getSession();
    LoggedUser user = new LoggedUser(UserId, UserPassword,role);
    session.setAttribute("loggedUser",user);
    // inoltra la risposta verso un altro servlet o pagina JSP
    // page = ricava la pagina dalla request invocata dal client oppure in altro modo
    res.sendRedirect(res.encodeRedirectURL(page));
  }
}
```

LoggedUser è una classe particolare utilizzata in MokaPackages ([MBPACK]), il framework di MokaByte il quale implementa una struttura simile a Struts: in questo caso LoggedUser permette anche di memorizzare il ruolo dell'utente loggato e quindi consente all'interno del metodo service della servlet di controllo di verificare se, ad esempio, un'azione associata all'URL add-item-to-cart.run è eseguibile dall'utente in questione. In questo caso ogni azione deve essere definita associando un ruolo.

In MBPACK la servlet che effettua il login in realtà è una Action Java mentre quella che effettua il controllo è un FilterServlet che filtra in modo trasparente tutte le invocazioni da parte del client.

Autenticazione basata su certificati

Nel caso in cui si desideri utilizzare un livello di sicurezza alto, si può ricorrere alla autenticazione di tipo DIGEST, basata sul meccanismo dello scambio di chiavi pubbliche e private (certificati).

Questo argomento è molto vasto e richiede una buona conoscenza di base delle tecniche e della teoria legata ai certificati, ai meccanismi di autenticazione e soprattutto a come il server utilizzato gestisce tali oggetti.

Non si entrerà nel dettaglio di questi aspetti, trattandosi argomenti non strettamente legati al mondo della programmazione delle servlet. Inoltre, similmente alla autenticazione BASIC e FORM, molti dettagli legati al funzionamento dei certificati digitali e al meccanismo di autenticazione DIGEST sono legati non tanto al servlet container, ma piuttosto al server HTTP utilizzato a fronte del container.

Una tipica configurazione di produzione vede Apache HTTP Server utilizzato in abbinamento con Jakarta Tomcat: il primo svolge tutte le operazioni di interfacciamento verso il client web fra cui anche l'autenticazione tramite certificati digitali. È infatti preferibile utilizzare per questo lavoro gli strumenti di configurazione e le funzionalità del server Apache piuttosto che basare tutto il lavoro su Tomcat, sia per motivi di prestazioni che per il maggior supporto offerto dal server HTTP.

Normalmente, in tale scenario Tomcat riceve, se il connettore lo permette, le informazioni di autenticazione inoltrate dal server HTTP posto come front end verso il client.

Nel caso in cui si decida di configurare direttamente il servlet container per l'autenticazione DIGEST, si dovrà modificare il file web.xml cambiando la parte relativa al tag <login-config>

```
<login-config>
  <auth-method>CLIENT-CERT</auth-method>
</login-config>
```

Inoltre si può forzare a utilizzare il protocollo SSL per il trasporto dei dati modificando un pezzo del blocco <security-constraint> di web.xml

```
<user-data-constraint>
  <transport-guarantee>
    INTEGRAL
  </transport-guarantee>
</user-data -constraint>
```

In questo caso il valore INTEGRAL garantisce che i dati ricevuti siano integri ovvero che non siano stati modificati durante il tragitto. L'altro valore consentito, CONFIDENTIAL, garantisce che i dati non siano stati letti da una terza parte non autorizzata e di fatto implica anche che i dati siano integri.

In genere in questi casi è il server che decide come e quando effettuare tutt i controlli necessari per garantire tali livelli di protezione.

Una applicazione web basata su servlet o pagine JSP può effettuare i controlli necessari accedendo a tali informazioni tramite i metodi messi a disposizione dalla API. Ad esempio può stabilire se la connessione è sicura tramite il metodo

```
public boolean ServletRequest.isSecure()
```

che restituisce il valore true se il server reputa che la connessione con il client rispettia i requisiti minimi di sicurezza. Valutare se e come la connessione sia sicura dipende da molti fattori, fra cui

stranamente fino a poco tempo fa anche la localizzazione geografica dato che algoritmi di protezione a 128 bit non potevano essere esportati dagli USA.

Con gli attributi di sicurezza javax.servlet.request.cypher_suite e javax.servlet.request.key_size si possono avere informazioni sull'algoritmo utilizzato e sulla dimensione delle chiavi usate per crittografare i dati.

Quando un client viene autenticato in modalità CLIENT-CERT, la servlet può utilizzare il metodo getUserPrincipal() per ricavare il nome dell'utente ricavandolo dal campo Distinguished Name del certificato.

La servlet può ricavare il certificato dalla request e operare tutti i controlli applicativi del caso:

```
java.security.cert.X509Certificate cert = (java.security.cert.X509Certificate)
                                    request.getAttribute("java.security.cert.X509Certificate");
```

Se il server supporta J2SE1.2 (JDK 1.2), allora in questo caso si otterrà un oggetto java.security.cert.X509Certificate che rappresenta un certificato X509v.

La API 2.3: i filtri

Con la versione 2.3 della API sono state introdotte alcune soluzioni volte a migliorare ulteriormente la modularità sia dell'application server che delle web application installate. Fra le molte modifiche, ogni web application utilizza un classloader e uno spazio di indirizzamento separati rispetto a quelli del container. In questo modo si può evitare fra le altre cose l'insorgere di conflitti fra il parser XML utilizzato da Tomcat per interfacciarsi con i vari file di configurazione XML e quelli eventualmente utilizzati dalle web application.

Modifiche alla Servlet API 2.3

Ecco un breve elenco delle innovazioni apportate alla Servlet API 2.3:

- definizione ufficiale del JDK 1.2 o successivi come piattaforma di base per l'esecuzione di servlet;
- introduzione del concetto di filtro;
- nuovo ciclo di vita delle servlet (più in linea con la filosofia tipica di J2EE, imitando ad esempio ciò che avviene in EJB) e contemporanea introduzione di nuovi eventi legati ai vari passaggi di stato o cambiamenti nel context;
- nuovo supporto per l'internazionalizzazione;
- formalizzazione del meccanismo di dipendenza fra i vari file JAR;

- formalizzazione del meccanismo di class loading;
- nuovi attributi per la gestione degli errori e della sicurezza;
- sostituzione della classe HTTPUtils;
- aggiunta di numerosi metodi alle varie classi;
- migliore definizione dell'interfacciamento con i vari DTD XML.

Pre- e postprocessamento di HTTP

Da sempre una delle caratteristiche più interessanti della tecnologia delle web application è quella che consente di mettere in comunicazioni risorse differenti: si pensi ad esempio alla possibilità di effettuare forward da una risorsa dinamica a un'altra, alla capacità di condividere oggetti di contesto fra una pagina JSP e una servlet e viceversa, o alla creazione di catene di servlet in esecuzione in cascata. Il panorama è quanto mai variegato e in continua evoluzione: gli sviluppi sono praticamente illimitati, se si pensa alla possibilità di mescolare le normali funzionalità che un web server mette a disposizione (alias, url dinamici, reindirizzamenti) con le varie tecniche utilizzabili all'interno di una web application.

Con la nuova Servlet API 2.3 e l'introduzione del concetto di filtro, un nuovo strumento si aggiunge alle tecniche già note.

Formalmente un filtro si definisce come un preprocessore di una request HTTP prima che questa raggiunga la servlet, e postprocessore della risposta prima che questa venga inviata al client. Un filtro può effettuare le seguenti operazioni:

- intercettare una chiamata a una servlet prima che la chiamata sia passata alla servlet e da questa processata;
- modificare la sezione degli header dei vari pacchetti HTTP prima del loro inoltro alla servlet;
- modificare la sezione degli header dei vari pacchetti HTTP prima della spedizione al client;
- intercettare una chiamata di una servlet dopo che tale servlet sia stata invocata.

La prima cosa che viene in mente è realizzare un filtro che controlli tutto il traffico diretto verso le servlet installate su un server per effettuare il *trace* del flusso dati (magari con relativo log su file).

Parallelamente si potrebbe immaginare il caso in cui tali dati siano effettivamente modificati prima di essere inviati alla servlet o al client: fra gli esempi presenti nella distribuzione 4.1.x di

Tomcat, vi è un filtro che permette di effettuare una compressione zip/gzip dei dati se il flusso supera una determinata soglia.

Un altro caso interessante potrebbe essere quello di introdurre sistemi di reindirizzamento in modo dinamico da e verso tutte le chiamate verso determinate servlet. Si pensi al caso presentato nel paragrafo relativo al meccanismo di login custom, in cui si controlla la presenza di un oggetto LoggedUser all'interno della sessione utente.

Un altro tipico utilizzo dei filtri è quello che permette di intercettare le varie richieste e di redirezionare il flusso della applicazione in modo dinamico. Si immagini il caso in cui una o più applicazioni devono collegarsi verso un altro sito esterno: se questo sito esterno dovesse per una qualche ragione non rispondere alle invocazioni o modificasse la sua interfaccia di invocazione (cambia l'URL per invocare un servizio di qualche tipo), si dovrebbe intervenire in modo pesante e in molti punti per poter modificare tutti i link, le servlet o le pagine JSP delle applicazioni interne.

Un semplice filtro potrebbe effettuare questo lavoro in modo trasparente, inviando un messaggio all'utente, modificando i parametri di invocazione al volo o, nei casi peggiori, effettuando una redirect su una pagina di errore appositamente creata, evitando contemporaneamente che le servlet tentino di collegarsi al sito bloccato.

Anche se gli esempi mostrati possono essere realizzati con strumenti alternativi ai filtri (il servlet chaining o tramite le regole di reindirizzamento configurate nel server HTTP), l'utilizzo dei filtri rende le cose molto più semplici, eleganti e soprattutto simili con il resto della logica delle web applications di J2EE.

I filtri

La prima cosa che è necessario fare per poter realizzare un filtro è implementare l'interfaccia javax.servlet.Filter che definisce i seguenti tre metodi

```
void init(FilterConfig config)
FilterConfig getFilterConfig ()
void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
```

Il primo esegue l'inizializzazione del filtro, mentre getFilterConfig() consente di ricavare dall'esterno l'oggetto di configurazione del filtro. All'interno di doFilter() verranno infine eseguite le operazioni di filtro. L'interfaccia FilterConfig mette a disposizione alcuni metodi che consentono fra l'altro di ottenere il nome del filtro e i parametri di inizializzazione, prelevando tali valori direttamente dal file XML di configurazione, in modo analogo al caso delle servlet standard. In maniera simile ai metodi doGet(), doPost() e service() delle servlet, il metodo doFilter() riceve i parametri corrispondenti alla invocazione da parte del client e alla risposta da inviare nel canale HTTP: gli oggetti ServletRequest e ServletResponse hanno lo stesso significato che nel caso delle servlet.

È possibile definire una catena di filtri in grado di effettuare le loro modifiche in sequenza per la stessa invocazione/risposta: l'oggetto FilterChain rappresenta proprio tale catena. Per fare questo dentro il metodo doFilter() si deve scrivere

```
chain.doFilter(request, response)
```

Se invece non si passa il controllo al filtro successivo, la chiamata verrebbe interrotta e non inoltrata alla servlet o al client (se si volesse ad esempio bloccare le comunicazioni).

Ecco un semplice esempio che mostra come calcolare il tempo di esecuzione di una ipotetica risorsa sul server per ogni invocazione da parte di un client.

```
public class TimeFilter implements Filter{
  FilterConfig config;
  ServletContext context;

  // esegue l'inizializzazione del filtro
  public void init(FilterConfig config){
    this.config = config;
    context=config.getServletContext() ;
  }

  public FilterConfig getFilterConfig(){
    return this.config;
  }

  public void doFilter(ServletRequest req, ServletResponse res,
                        FilterChain chain) throws ServletException, IOException {
    long InvocationTime=System.currentTimeMillis();
    chain.doFilter(req, res);
    long EndExecutionTime=System.currentTimeMillis();
    String Message;
    Message="Tempo di esecuzione della richiesta: ";
    Message+=""+(EndExecutionTime-InvocationTime);
    context.log(Message);
    . . .
  }
}
```

In questo semplice esempio, il metodo doFilter() verrà invocato in occasione di chiamate da parte di un client, e andrà a scrivere su file di log il tempo che è stato necessario per effettuare l'invocazione. Come si può notare, la cosa interessante è che il passaggio all'elemento successivo nella catena è effettuato in mezzo alla chiamata di doFilter(): in questo modo il controllo ritornerà al filtro chiamante, dando vita a un vero e proprio annidamento delle chiamate.

Wrapping delle comunicazioni

Un'altra interessante caratteristica offerta con la nuova API è la possibilità di modificare le risposte e le invocazioni inglobando gli oggetti ServletRequest e ServletResponse in appositi wrapper al fine di personalizzarne il contenuto. Gli oggetti WrapperServletRequest e WrapperServletResponse sono stati introdotti proprio a questo scopo. Per comprenderne il funzionamento si potrebbe

prendere spunto da uno degli esempi presenti in Tomcat 4. In questo caso, un filtro intercetta le chiamate e le risposte ed effettua una compressione dei dati se il pacchetto da trasferire supera complessivamente una determinata dimensione. Ecco un breve estratto del metodo doFilter():

```
public void doFilter(ServletRequest req, ServletResponse res,
                        FilterChain chain) throws ServletException, IOException {

  // esegue una serie di controlli per verificare se il client
  // supporta la compressione dei dati
  . . .
  // adesso nel caso in cui il traffico sia in risposta
  // effettua la compressione dei dati
  if (res instanceof HttpServletResponse) {
    // si utilizza un Wrapper di risposta personalizzato che permette
    // la compressione dei dati
    CompressionServletResponseWrapper wrRes;
    wrRes = new CompressionServletResponseWrapper((HttpServletResponse)res);
    // imposta la soglia oltre la quale i pacchetti verranno compressi
    wrRes.setCompressionThreshold(compressionThreshold);
    try {
      // inoltra la chiamata utilizzando il wrapper di risposta
      chain.doFilter(request, wrappedResponse);
    }
    finally {
      wrappedResponse.finishResponse();
    }
  return;
  }// fine metodo doFilter()
}// fine classe
```

Si noti come la risposta venga intercettata e successivamente reindirizzata verso gli altri filtri della catena, provvedendo ad inglobare la risposta in un wrapper apposito: in questo caso si tratta di un oggetto CompressionServletResponseWrapper che provvede a fornire i metodi necessari per la compressione dei dati. Non potendo riportare completamente il codice di tale classe (sia per motivi di spazio che per imposizioni contrattuali) se ne possono brevemente analizzare i punti più interessanti. Per prima cosa la definizione della classe che deve estendere HttpServletResponseWrapper

```
public class CompressionServletResponseWrapper extends HttpServletResponseWrapper {
```

Il costruttore riceve dall'esterno una generica HttpServletResponse di cui effettuerà poi la funzione di wrapping

```
public CompressionServletResponseWrapper(HttpServletResponse response) {
  super(response);
  origResponse = response;
}
```

Il metodo createOutputStream() crea uno stream di compressione, classe CompressionResponseStream, sulla base della risposta originaria

```
public ServletOutputStream createOutputStream() throws IOException {
  return (new CompressionResponseStream(origResponse));
}
```

Infine uno dei metodi deputati alla scrittura delle informazioni nel canale HTTP si appoggia ai metodi di compressione offerti dalla CompressionResponseStream

```
public void flushBuffer() throws IOException {
  ((CompressionResponseStream)stream).flush();
}
```

Le operazioni di compressione vere e proprie verranno effettuate all'interno della classe CompressionResponseStream che, utilizzando alcune classi e metodi del package java.util.zip, trasforma lo stream dei dati in pacchetti compressi (si veda la bibliografia).

Installazione e deploy dei filtri

Il deploy di un filtro è un'operazione molto semplice e segue fedelmente la procedura da utilizzare per il deploy di una comune web application.

Per prima cosa si deve ricreare una struttura a directory tipica delle web application (basata sulle directory WEB-INF, classes, META-INF etc..) e posizionare i file .class nella directory webapproot\WEB-INF\classes.

A questo punto nel file web.xml si dovrà procedere alla definizione del filtro, dei suoi parametri di inizializzazione e degli eventuali alias URL. Ecco un esempio

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">

  <web-app>
    <servlet>
      <servlet-name>filteredservlet</servlet-name>
      <servlet-class>com.mokabyte.servlet.TestServlet</servlet-class>
    </servlet>
    <servlet-mapping>
      <servlet-name>filteredservlet</servlet-name>
      <url-pattern>/filteredservlet</url-pattern>
    </servlet-mapping>
    <filter>
      <filter-name>
        logtime
      </filter-name>
```

```
    <filter-class>
      ServletFilter
    </filter-class>
    <init-param>
      <param-name>maxtime</param-name>
      <param-value>212</param-value>
    </init-param>
    </filter>
    <filter-mapping>
      <filter-name>logtime</filter-name>
      <url-pattern>/servlet/FilteredSerlvet</url-pattern>
    </filter-mapping>
  </web-app>
```

Nel codice evidenziato è stato definito un filtro indicandone il nome logico, la classe corrispondente e alcuni parametri di inizializzazione. Successivamente, tramite il tag <filter-mapping> si dice al server HTTP quando tale filtro dovrà essere eseguito. In questo caso il controllo verrà intercettato solo in occasione dell'invocazione della servlet di prova FilteredServlet. Se invece si fosse scritto

```
<filter-mapping>
  <filter-name>logtime</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

il filtro avrebbe filtrato le chiamate per tutte le risorse mappate sotto la web root, ovvero anche file HTML statici, pagine JSP, programmi CGI-BIN.
Si noti la sezione

```
<init-param>
  <param-name>maxtime</param-name>
  <param-value>212</param-value>
</init-param>
```

che permette di passare dei parametri di inizializzazione al filtro: tali parametri saranno recuperati all'interno del metodo init(). Ad esempio

```
public void init(FilterConfig filterConfig) throws ServletException {
  this.filterConfig = filterConfig;
  this.attribute = filterConfig.getInitParameter("maxtime ");
}
```

Si noti infine la presenza nel file web.xml della prima riga

```
< . . . "http://java.sun.com/dtd/web-app_2_3.dtd">
```

necessaria per permettere l'utilizzo dei nuovi tag <filter> <filter-mapping> e simili non presenti nella versione 2.2.

Gli ascoltatori e ciclo di vita

Il ciclo di vita di una servlet segue precise regole: a partire dalla inizializzazione tramite il metodo init(), la gestione delle richieste dei client con il metodo service(), e lo spegnimento della servlet tramite il metodo destroy().

Da quando, con la nascita della piattaforma Java2EE, Sun ha introdotto concetti come container e servlet context, lo scenario si è notevolmente complicato: il servlet context infatti permette a due servlet o a una pagina JSP e una servlet di scambiarsi oggetti, oppure a una servlet di ottenere maggiori informazioni circa l'oggetto chiamante e i parametri di sessione. Anche l'oggetto Session, presente fin dalla prima release delle Servlet API, ha assunto un ruolo molto più importante, specie nel caso di web application basate su pagine JSP e servlet.

Per questo motivo, al fine di permettere una maggiore e più semplice gestione di tutte queste funzionalità, è stato aggiunto un utilissimo meccanismo basato su ascoltatori sintonizzati sui vari cambiamenti che interessano sia il servlet context che l'oggetto Session.

Implementando alcune semplici interfacce, un ascoltatore adesso potrà essere informato tutte le volte che un attributo viene aggiunto al contesto o alla sessione. Più precisamente, mentre la sessione viene associata ad ogni client collegato, il context nasce e muore insieme a una web application. Visto quindi che l'ascoltatore può essere notificato sugli eventi di creazione e distruzione del servlet context, non è improprio, anche se formalmente non corretto, dire che le modifiche apportate hanno dato vita a un nuovo ciclo di vita della servlet: in realtà niente è cambiato nel ciclo di vita. Il meccanismo di notifica degli eventi è lo stesso degli eventi utilizzati in Swing o AWT, meccanismo che come noto è basato sul cosiddetto Delegation Model [3], o pattern Observer[4].

Ascoltatori di contesto

L'interfaccia che deve essere implementata per creare un ascoltatore sincronizzato sul ciclo di vita del servlet context è la ServletContextListener, che contiene i seguenti due metodi:

```
void contextInitialized(ServletContentEvent sce)
void contentDestroyed(ServletContentEvent sce)
```

Il primo viene richiamato quando la web application è pronta per eseguire le chiamate dei client HTTP. In particolare la web application non sarà pronta fino a che questo metodo non abbia terminato la sua esecuzione. Il secondo metodo invece viene invocato dal server nel momento in cui la web application viene spenta, o quando più genericamente il context viene rimosso (ad esempio nel caso di applicazioni mobili fra più application server). La gestione delle richieste viene sospesa subito prima della invocazione di tale metodo.

In entrambi i casi, l'oggetto ServletContextEvent, tramite il metodo getServletContext(), permette di ricavare un riferimento al contesto appena dopo essere stato inizializzato o prima di essere rimosso.

Nel caso in cui un oggetto sia interessato a osservare il movimento degli oggetti inseriti o rimossi dal contesto, potrà implementare l'interfaccia ServletContextAttributeListener. In questo caso i metodi della interfaccia da implementare saranno:

```
void attributeAdded(ServletContextAttributeEvent sceae)
void attributeRemoved(ServletContextAttributeEvent sceae)
void attributeReplaced(ServletContextAttributeEvent sceae)
```

Il significato e funzionamento di tali metodi dovrebbe apparire piuttosto intuitivo. L'oggetto ServletContextAttributeEvent, che deriva direttamente dal ServletContextEvent estende il padre aggiungendo i metodi getName() e getValue() i quali permettono di avere maggiori informazioni sugli oggetti rimossi, aggiunti o modificati.

Ascoltatori di sessione

Per quanto riguarda la creazione di ascoltatori sincronizzati sugli eventi delle sessioni, il meccanismo è piuttosto simile. Anche in questo caso è stata aggiunta una interfaccia, la HTTPSessionListener, che mette a disposizione i seguenti metodi

```
void sessionCreated(HttpSessionEvent hse)
void sessionDestroyed(HttpSessionEvent hse)
```

Il primo viene invocato al momento della creazione della sessione, mentre l'altro alla sua distruzione. Anche in questo caso gli oggetti passati come parametri permettono di ottenere maggiori informazioni circa la sessione in esame. In particolare, il metodo getSession() consente di ottenere un oggetto Session.

Anche in questo caso è possibile implementare un controllo più fine concentrando l'attenzione di un ascoltatore solo sugli oggetti aggiunti o rimossi dalla sessione: infatti implementando l'interfaccia HttpSessionAttributeListener e i metodi

```
void attributeAdded(HttpSessionBindingEvent hsbe)
void attributeRemoved(HttpSessionBindingEvent hsbe)
void attributeReplaced(HttpSessionBindingEvent hsbe)
```

l'oggetto HttpSessionBindingEvent estende la più generica HttpSessionEvent, e aggiunge i metodi getName() e getValue() per conoscere nome e valore dell'oggetto inserito, rimosso o modificato nella sessione. La scelta del nome al posto di HttpSessionAttibuteEvent è stata fatta per mantenere la compatibilità con le versioni precedenti della API, visto che era già presente una classe con questo nome. È possibile che nelle versioni successive, Sun decida di apportare modifiche in tal senso.

Implementazione degli ascoltatori

In questo paragrafo viene mostrato come implementare un ascoltatore di contesto e uno di sessione, cercando successivamente di dare un senso e una motivazione sull'utilizzo di questi strumenti.

Il primo passo per realizzare un ascoltatore è creare una classe che implementi una delle interfacce appena viste. Si consideri prima il caso di un ascoltatore di contesto: la classe potrebbe avere questa struttura

```
public final class MyContextListener implements ServletContextAttributeListener, ServletContextListener {
    . . .
}
```

In questo caso l'ascoltatore preso in esame sarà in grado di "ascoltare" sia gli eventi relativi al contesto vero e proprio che ai relativi attributi.

I vari metodi dell'interfaccia potrebbero in una versione molto semplice di questo ascoltatore effettuare un log su file tutte le volte che si verifica un evento. Ad esempio si potrebbe scrivere

```
public void attributeAdded(ServletContextAttributeEvent event) {
  fileLog("Aggiunto attributo al contesto " + event.getName() + ", " + event.getValue());
}
```

dove il metodo `fileLog()` utilizza le funzionalità messe a disposizione dal contesto per effettuare log su file; ad esempio

```
private void fileLog(String message) {
  if (context != null)
    context.log("MyContextListener: " + message);
  else
    System.out.println("MyContextListener: " + message);
}
```

In scenari di produzione reali si consiglia l'utilizzo di sistemi di log più solidi e potenti, come il Log4J del progetto Jakarta [Log4J] o quello introdotto con il JDK 1.4.

Gli altri metodi dell'ascoltatore potrebbero effettuare operazioni analoghe o del tutto differenti. In questo caso si tralasciano questi aspetti rimandando alla fantasia del programmatore.

Per la realizzazione di un ascoltatore di sessione invece si potrebbe creare un oggetto che, come nel caso precedente, possa funzionare come ascoltatore di tutti i tipi di eventi riguardanti una sessione. Ad esempio

```
public final class MySessionListener implements HttpSessionAttributeListener, HttpSessionListener {
  // uno dei metodi per la gestione degli eventi
  public void attributeAdded(HttpSessionBindingEvent event) {
    fileLog("Attributo aggiunto alla sessione " + event.getSession().getId() +  ": "
             + event.getName() + ", " + event.getValue() );
```

```
  }
}
```

Riconsiderando per un momento il meccanismo implementato in Swing o AWT circa la gestione degli eventi, si ricorderà che per rendere operativo un ascoltatore lo si deve registrare come listener di una determinata sorgente. In questo caso il servlet container effettua una operazione del tutto analoga: l'unica differenza è che essendo la sorgente unica per tutti i listener la registrazione si limita a comunicare al server il nome della classe associata ad un listener. Per fare questo al solito si ricorre a un apposito tag XML, da inserire nel file web.xml della web application che contiene l'oggetto ascoltatore.
Riconsiderando per un momento iAd esempio si potrebbe pensare di scrivere:

```
<!—un semplice ascoltatore di eventi di contesto -->
  <listener>
    <listener-class>com.mokabyte.listeners.MyContextListener</listener-class>
  </listener>
```

Come e quando utilizzare gli ascoltatori

Riconsiderando per un momento il significato di questa importante aggiunta alla API potrebbe non essere immediatamente chiaro a tutti, ma in realtà rappresenta un ulteriore passo in direzione della semplificazione della gestione delle applicazioni Enterprise. In tal senso si potrebbero elencare moltissimi casi in cui l'utilizzo degli ascoltatori può essere utile: per brevità si analizzeranno solo due semplici casi.

Il primo riguarda la gestione della connessione verso un database, e la sua ottimizzazione tramite l'utilizzo di ascoltatori del contesto. In questo caso si potrebbe utilizzare un connection pool al fine di ottimizzare le risorse e ridurre i tempi di attesa nella apertura e chiusura delle connessioni. Tale connection pool potrebbe essere inizializzato (lettura dei parametri di configurazione, apertura delle connessioni, e così via) prima del suo utilizzo, tramite un metodo startup(), e viceversa disattivato quando non più necessario, con il metodo shutdown().

In questo scenario si dovrebbe consentire ad ogni servlet di ricavare una propria istanza di connection pool, ma anche di effettuare l'operazione di inizializzazione una sola volta, alla prima richiesta da parte di una servlet.

Tipicamente questo tipo di operazioni viene svolto tramite l'utilizzo combinato del pattern Factory, per permettere ad ogni servlet di ricavare una propria istanza del connection pool (vedi [FACT]) e del Singleton, per effettuare l'inizializzazione una volta soltanto. Questa soluzione non è particolarmente complessa, ma rende necessario inserire nei metodi init() e destroy() di ogni servlet una serie di operazioni concettualmente non banali.

Si potrebbe pensare di sfruttare il ciclo di vita del contesto tramite opportuni ascoltatori per effettuare una inizializzazione universale per tutte le servlet. Ad esempio

```
public void contextInitialized(ServletContextEvent event) {
  // effettua la creazione del connection pool
  // invocando il costruttore con gli opportuni parametri
```

```
    ConnectionPool cp= new ...
    cp.startup();
    event.getServletContext().setAttribute("connectionpool",cp);
  }
```

A questo punto dopo la creazione del contesto, tutte le servlet potranno ricavare il connection pool che sarà disponibile come attributo di sistema. Ad esempio:

```
  public void doGet(HttpServletRequest req, HttpServletResponse res)
              throws ServletException, IOException{
    // ricava il connectio pool come attributo di contesto
    ServletContext context = getServletContext();
    ConnectionPool cp=(ConnectionPool)context.getAttribute("connectionpool");
    con=cp.getConnection();
  }
```

Anche la gestione dello spegnimento del connection pool in modo corretto, ovvero quando tutte le servlet verranno rimosse dalla memoria, potrà essere eseguita da un ascoltatore opportunamente sincronizzato:

```
  public void contextDestroyed(ServletContextEvent event) {
    // ricava il connectio pool come attributo di contesto
    ServletContext context = event.getServletContext();
    ConnectionPool cp=(ConnectionPool)context.getAttribute("connectionpool");
    cp.shutdown() ;
  }
```

Un altro interessante utilizzo dei listener, sia di sessione che di contesto, potrebbe essere quello di realizzare un sistema monitor che controlli in ogni momento il numero di web application in funzione e di sessioni attive, ovvero di client connessi. Non si entrerà nei dettagli di una applicazione di questo tipo, visto che lo spazio a disposizione non lo permette. Invece per comprendere meglio quello che potrebbe essere il risultato finale, si può vedere in fig. 5.9 l'output prodotto dalla web application di gestione remota di Tomcat 4.0.

Altre innovazioni nella API 2.3

Di seguito sono riportate alcune delle modifiche o aggiunte minori introdotte con la nuova API.

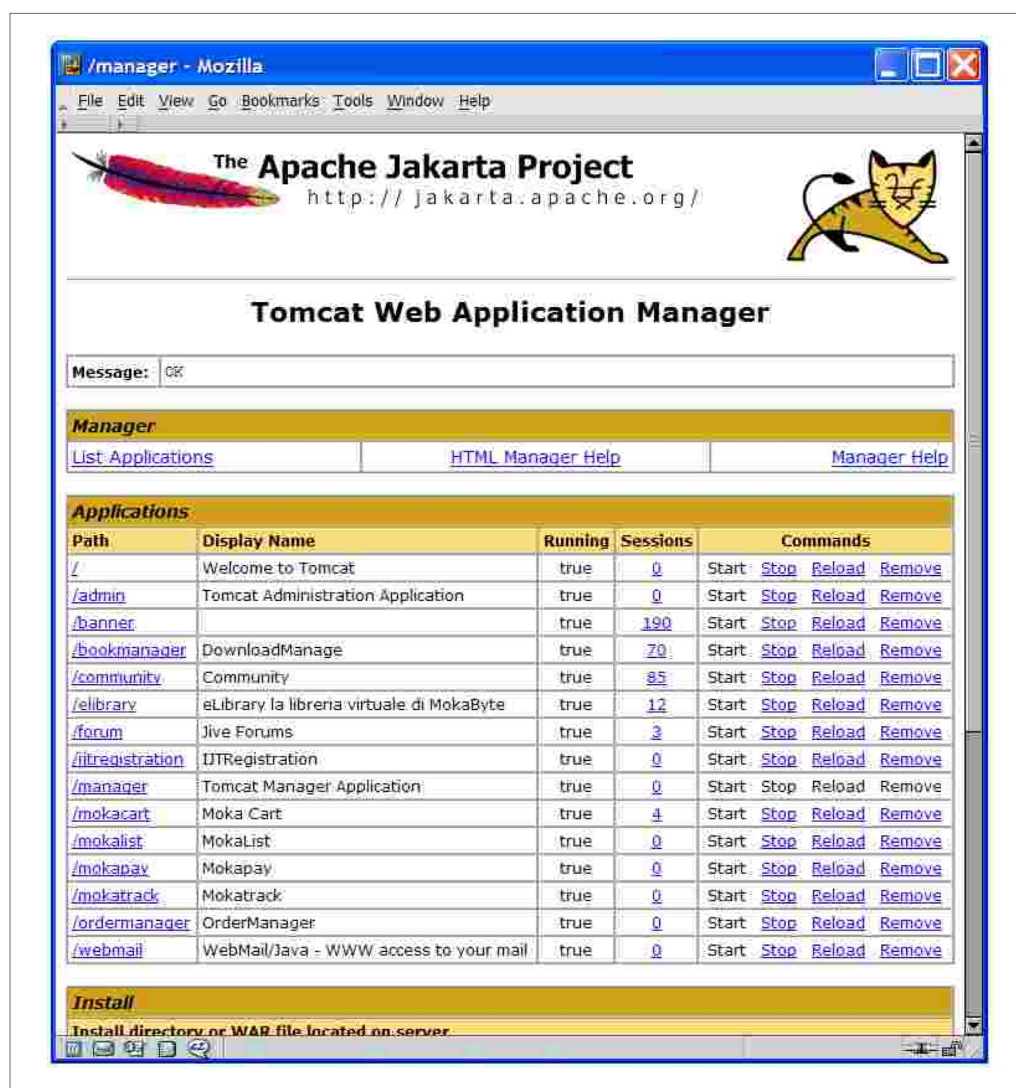
Selezione della codifica dei caratteri

Con la versione 2.3 della API è possibile adesso specificare il set di caratteri con cui la servlet deve interpretare la request del client. Il metodo

```
  request.setCharacterEncoding(String encoding)
```

consente di effettuare tale impostazione e garantendo ad esempio una successiva lettura dei parametri di invocazione. Ad esempio se il client utilizza il set di caratteri Shift_JIS, uno dei set di

Figura 5.9 – *La web application Manager fornita in Tomcat 4.0 consente di monitorare ogni singola web application in funzione e il numero di sessioni attive. Le applicazioni in elenco sono alcune delle web application utilizzate per il sito MokaByte, monitorate in un momento di basso traffico sul sito (si notino le sessioni attive).*



caratteri utilizzati per la lingua giapponese, per poter correttamente interpretare un parametro passato all'invocazione, si potrebbe pensare di scrivere

```
request.setCharacterEncoding("Shift_JIS");
String param=req.getParameter("param");
```

Questo genere di operazioni è necessario in tutti quei casi in cui il set di caratteri di default, il Latin-1 (ISO 8859-1) può portare a errori di interpretazione, come nel caso della lingua giapponese appunto. Normalmente infatti il set di caratteri, se diverso da quello di default, dovrebbe essere specificato come intestazione nell'header della richiesta, ma non è detto che tale assunzione sia sempre rispettata dai vari browser o client HTTP.

Per questo motivo il set di caratteri deve essere specificato prima di ogni operazione di tipo getParameter() o getReader(); la chiamata al metodo setCharacterEncoding() può generare una eccezione di tipo java.io.UnsupportedEncodingException se la codifica non è supportata.

Gestione separata dei vari classloader

Una modifica piuttosto importante è stata apportata al classloader delle web applications. Adesso ogni web application esegue il caricamento delle classi in uno spazio di memoria indipendente dalle altre, ma anche e soprattutto indipendente da quello del server. Questo secondo aspetto, apparentemente non rilevante, ha invece importanti ripercussioni in fase di deploy di una applicazione. Infatti prima non era infrequente che una qualche libreria caricata dalla JVM del server andasse in conflitto con quelle necessarie alle varie web application, a discapito di queste ultime (dato che il server carica in memoria le classi prima delle singole web application). Il caso più eclatante e frequente era quello del conflitto di parser XML: dato che il server spesso utilizza file XML per la configurazione complessiva, eso usa strumenti come Xerces o simili, che spesso però sono impiegati anche all'interno della web application, magari con una versione differente.

È per questo motivo che infatti dalla versione 4.0 Tomcat non fornisce più nessun parser XML di default alle varie web application le quali devono provvedere a contenerne uno in modo autonomo, ad esempio tramite un file .jar all'interno della directory lib dell'applicazione.

Nuovi attributi di protezione

Nell'ambito delle connessioni in modalità protetta tramite protocollo HTTPS, la nuova API fornisce due nuovi attributi in grado di determinare il livello di sicurezza relativamente alla chiamata in atto. I due attributi sono

```
javax.servlet.request.cipher_suite
javax.servlet.request.key_size
```

il primo permette di ricavare la *cipher suite* utilizzata da HTTPS se presente, mentre il secondo indica il numero di bit utilizzati dall'algoritmo di crittografia. Utilizzando tali informazioni una servlet adesso può rifiutare una connessione se il livello di sicurezza o l'algoritmo utilizzato non risultano idonei. Ad esempio

```
public void service(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {
  Integer CriptSize =(Integer) req.getAttribute("javax.servlet.request.key_size");
  if(CriptSize == null || CriptSize.intValue() < 128){
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<head><title>HTTPS Servlet </title></head>");
    out.println("<body>");
    out.println("<p> La chiamata non è sufficientemente sicura e verrà ignorata </p>");
    out.println("</body></html>");
  }
}
```

Altre modifiche minori

Fra le modifiche di minore importanza, si può fare riferimento ai metodi della classe HttpUtils, che sono stati spostati adesso in altri punti secondo una logica più consona con l'intera API. I metodi relativi alla creazione di URL a partire da una request sono stati spostati nella HTTPServletRequest stessa, cui contemporaneamente sono stati aggiunti altri metodi di utilità. Ad esempio

```
StringBuffer request.getRequestUrl()
```

restituisce una StringBuffer contenente l'URL della richiesta. Il metodo

```
java.util.Map request.getParameterMap()
```

restituisce una mappa non modificabile con tutti i parametri dell'invocazione. I nomi dei parametri sono utilizzati come chiave della mappa, mentre i valori sono memorizzati nei corrispondenti valori della mappa. Per i valori multipli, al momento non è stato specificato quale convenzione verrà utilizzata: probabilmente tutti i valori verranno restituiti come array di stringhe.

Inoltre sono stati aggiunti all'oggetto ServletContext i metodi getServletContextName(), che consente di ricavarne il nome, e getResourcePaths() che restituisce un elenco non modificabile di stringhe corrispondenti a tutti i path delle risorse disponibili all'interno del contesto. In particolare ognuna di queste stringhe inizia per "/" e deve essere considerata un indirizzo relativo al contesto in esame.

All'oggetto HttpServletResponse invece è stato aggiunto il metodo resetBuffer() il cui scopo è quello di eliminare ogni contenuto all'interno del buffer, senza però cancellare le intestazioni o il codice di stato, prima che la risposta sia inviata al client. In caso contrario viene generata un'eccezione di tipo IllegalStateException.

Infine, dopo una lunga discussione del comitato che segue l'evoluzione della API, è stato chiarito, rispetto alla versione precedente che lasciava in sospeso questo aspetto, che il metodo HttpServletResponse.sendRedirect(String uri) effettui un reindirizzamento verso l'indirizzo specifica-

to traducendo tale indirizzo relativo in indirizzo assoluto ma relativamente al container e non al context in uso. Più semplicemente in una chiamata del tipo

```
response.sendRedirect("/index.html");
```

l'indirizzo relativo "index.html" verrà trasformato in indirizzo assoluto del tipo "http://server:port/index.html" e non "http://server:port/contextname/index.html".

Chi volesse ottenere il risultato opposto potrà utilizzare il metodo getContextName() preponendolo al nome dell'URI utilizzato.

Considerazioni sulla API 2.3

Questa nuova versione della Servlet API appare piuttosto interessante. Sicuramente le novità più importanti sono quelle relative all'introduzione del meccanismo dei filtri e dei listener. Questi due innovativi strumenti permetteranno un più agile e potente meccanismo di gestione di applicazioni web basate su tecnologia Servlet/JSP.

Per quanto riguarda invece le altre innovazioni, si tratta di migliorie di minor conto, ma che nel complesso mostrano come la API sia ormai giunta a un buon livello di maturità e stabilità; il tutto mostra come Sun creda nell'importanza di questo settore della tecnologia Java (insieme a JSP, EJB, Java&XML), e di fatto prosegua nel suo costante miglioramento e aggiornamento.

La Servlet API 2.4

La versione 2.4 della API è l'ultima disponibile al momento della revisione finale di questo capitolo: sul sito Sun è presente l'ultimo draft a cui si potrà fare riferimento per ulteriori approfondimenti (vedi [JSR154]).

Differentemente dalle release precedenti 2.2 e 2.3, in cui furono presentate importanti modifiche e aggiunte al framework (le self-contained web applications nella 2.2, i filtri e i listener nella 2.3), in questo caso non stati introdotti cambiamenti sostanziali. Per coloro che spesso si trovano a dover inseguire gli aggiornamenti tecnologici della piattaforma J2EE, questa notizia potrà apparire sicuramente positiva.

Questa nuova release dovrebbe rendere più semplice per i programmatori e per i produttori di servlet container il passaggio alla nuova versione della API. In sintesi ecco l'elenco delle novità introdotte con la 2.4:

- per poter utilizzare le servlet in API 2.4 adesso è necessario utilizzare l'HTTP/1.1 e la Java 2 Platform, Standard Edition versione 1.3, anche se si può utilizzare la versione 1.4 della J2EE;
- l'interfaccia ServletRequest fornisce adesso nuovi metodi che permettono di ricavare informazioni circa la connessione verso il client;

- un nuovo supporto per l'internazionalizzazione e per la scelta del set di caratteri;
- nuove funzionalità aggiunte alla RequestDispatcher;
- nuovi listener e nuovi metodi aggiunti alla ServletRequest;
- è stato deprecato l'utilizzo del modello sincronizzato basato sull'implementazione dell'interfaccia SingleThreadModel;
- è stata chiarita e definita la modalità di interazione con la HttpSession per quanto concerne la procedura di login;
- maggiore chiarimento del comportamento dei classloader e dei welcome-file;
- il file di descrizione del deploy, web.xml, adesso utilizza un XML Schema per la definizione della struttura e comprende nuovi elementi (tag) per la definizione di nuove funzionalità e comportamenti dell'applicazione.

Tutte queste modifiche, che potrebbero apparire molte e pesanti, in realtà assumono un peso diverso a seconda dei casi. Quello che si vedrà di seguito è una analisi dettagliata di ogni singolo aspetto cercando di comprendere anche quanto e come tali novità impattino realmente nella progettazione e implementazione di applicazioni web basate su servlet.

I nuovi requirements

Sicuramente la novità più importante da un punto di vista architetturale è data dal nuovo set di requirements necessari per poter utilizzare la nuova servlet API.

Una prima scelta piuttosto forte è stata quella relativa al protocollo HTTP: l'abbandono definitivo della versione 1.0, permette di sfruttare alcune importanti novità introdotte nella versione 1.1. La 1.0 rappresenta la versione originale del protocollo e fu standardizzata nel 1996. La 1.1 invece rappresenta la nuova versione del protocollo, la quale offre alcune importanti miglioramenti nelle performance, fra cui un più efficiente utilizzo delle connessioni HTTP, un miglior supporto per il client-side caching, la possibilità di utilizzare richieste HTTP multiple (*pipelining*), un più raffinato controllo sulla invalidazione della cache e modifiche alle regole di policy.

Questo passaggio rappresenta per certi versi una scelta piuttosto radicale anche perché attualmente non tutti i server (servlet container in particolare) supportano tale protocollo e sarà necessario un po' di tempo per aggiornare lo strato di comunicazione.

È stato aggiunto il nuovo codice di errore (status code 302) tramite la variabile statica

```
HttpServletResponse.SC_FOUND
```

dove *Found* è il nome utilizzato in HTTP/1.1 corrispondente a quello che in HTTP/1.0 era *Moved temporarily*. Il codice

HttpServletResponse.SC_MOVED_TEMPORARILY

continua a esistere e rappresenta il codice 302, ma è preferibile utilizzare il SC_FOUND: SC_MOVED_TEMPORARILY può essere considerato *deprecated*, anche se tecnicamente deprecare una variabile HTTP è impossibile.

Per quanto riguarda la JVM, ci sono novità anche in tal senso: con la 2.4 API il minimo necessario è rappresentato dalla versione 1.3 della J2SE. Formalmente poi è da notare che la nuova specifica verrà rilasciata nell'ambito della prossima J2EE 1.4: questo non significa che il container utilizzato debba offrire supporto completo per tutta la piattaforma Java 2 Enterprise Edition (ad esempio Tomcat non fornisce supporto per EJB o per JMS), ma nel caso ciò sia possibile, le servlet potranno trarre vantaggio dalla nuova piattaforma in modo più omogeneo e completo.

Nuovi metodi di richiesta

Sono stati aggiunti nuovi metodi alla classe ServletRequest per permettere al programmatore di ricavare maggiori informazioni circa il client chiamante. Questi metodi sono:

- getRemotePort(): restituisce il numero della porta IP del client o dell'ultimo proxy che ha inoltrato la richiesta;
- getLocalName(): restituisce l'hostname associato all'interfaccia di rete sulla quale la richiesta è stata ricevuta;
- getLocalAddr(): restituisce l'indirizzo IP dell'interfaccia di rete sulla quale la richiesta è stata ricevuta;
- getLocalPort(): restituisce l'IP port number dell'interfaccia di rete sulla quale la richiesta è stata ricevuta;

Questi metodi, il cui significato e funzionamento dovrebbe apparire piuttosto intuitivo, offrono uno strumento per agire a basso livello sullo strato di comunicazione con il client.

Da notare che alcuni metodi preesistenti sono stati aggiunti per offrire un maggior livello di informazioni: ad esempio i getServerName() e getServerPort() sono stati ridefiniti per esporre dettagli a livello HTTP, semplicemente restituendo la stringa "host:port" ricavata dall'host header HTTP.

Tutte queste informazioni possono essere particolarmente utili e importanti nel caso di sistemi bilanciati e di virtual hosting dato che permettono di ricavare un maggior numero di dettagli sul dove e sul come la servlet sia eseguita.

Internazionalizzazione

Con la nuova API, sono stati forniti alcuni nuovi metodi al fine di facilitare la gestione di contenuti localizzati. All'interfaccia ServletResponse sono stati aggiunti due nuovi metodi: il primo

è setCharacterEncoding(String encoding), il quale imposta la codifica dei caratteri da utilizzare per comporre la risposta. Questo metodo è utilizzabile in alternativa al classico setContentType(String) oppure alla impostazione del locale tramite il metodo setLocale(Locale). Come per tutti quei metodi e operazioni che hanno effetto sull'header della risposta HTTP, questo metodo non ha effetto se invocato dopo l'esecuzione della getWriter() o se una risposta è già stata inviata al client.

Il metodo getContentType(), invece, restituisce il content type della risposta, ovvero il charset impostato tramite i metodi setContentType(), setLocale() o setCharacterEncoding(), visti poco sopra. Se nessun content type è stato impostato, esso restituisce null.

Il metodo setCharacterEncoding() che si affianca al già presente getCharacterEncoding(), fornisce un modo più semplice per manipolare il charset della risposta, consentendo di evitare l'utilizzo ad esempio di una chiamata del tipo setContentType("text/html; charset=UTF-8").

Tutti questi metodi potrebbero apparire non troppo interessanti e utili, ma la loro presenza adesso costituisce un meccanismo uniforme per impostare il content type utilizzando congiuntamente i metodi setContentType(), setLocale() e setCharacterEncoding().

La scelta dell'utilizzo fra il metodo setLocale() o l'equivalente setCharacterEncoding() dipende dal contesto specifico. Brevemente si potrebbe dire che l'utilizzo di un oggetto Locale può essere più che sufficiente per la maggior parte dei casi, dato che implicitamente esso imposta anche il set di caratteri da utilizzare. Vi possono però essere alcuni casi in cui a uno stesso *locale* possono corrispondere diversi tipi di set di caratteri, o si desidera specificarli meglio.

Ad esempio, il *locale* giapponese potrebbe implicare l'utilizzo di due charset differenti Shift_JIS e EUC-JP. In questi casi, il metodo setCharacterEncoding() offre una granularità maggiore per specificare queste impostazioni.

In realtà la Servlet API 2.4 offre uno strumento ancora più flessibile per specificare il locale e il set di caratteri: tramite alcuni nuovi tag XML è adesso possibile mappare un determinato charset o locale, semplicemente editando il file di deploy web.xml. Ad esempio per specificare il charset Shift_JS da associare al locale ja si potrebbe scrivere

```
<locale-encoding-mapping-list>
  <locale-encoding-mapping>
    <locale>ja</locale>
    <encoding>Shift_JIS</encoding>
  </locale-encoding-mapping>
  <locale-encoding-mapping>
    <locale>zh_TW</locale>
    <encoding>Big5</encoding>
  </locale-encoding-mapping>
</locale-encoding-mapping-list>
```

dove ovviamente la coppia <locale-encoding-mapping> </locale-encoding-mapping> può essere ripetuta un numero arbitrario di volte quante sono le associazioni che si desidera specificare.

Nell'esempio visto si associa al locale ja il charset Shift_JIS e al zh_TW (Chinese/Taiwan) il charset Big5.

Nel momento in cui il diffondersi di client in grado di leggere e parlare con il charset UTF-8 dovesse crescere, si potrà modificare il tipo di risposta prodotta dall'applicazione senza dover modificare una sola riga di codice Java.

RequestDispatcher

Nel caso in cui si effettui una forward server-side, la servlet passa il controllo ad un altro URI, senza la necessità di inoltrare il controllo al browser.

Il metodo che normalmente si utilizza per questa operazione è il forward() della interfaccia RequestDispatcher.

I metodi getRequestURI(), getContextPath(), getServletPath(), getPathInfo() e getQueryString() restituiscono le informazioni relative all'URI invocato. A volte può essere necessario ricavare l'URI originale della richiesta, quello invocato prima della invocazione di forward(). Per questo sono stati aggiunti i seguenti attributi:

```
javax.servlet.forward.request_uri
javax.servlet.forward.context_path
javax.servlet.forward.servlet_path
javax.servlet.forward.path_info
javax.servlet.forward.query_string
```

Ad esempio, tramite l'istruzione

```
request.getAttribute("javax.servlet.forward.request_uri");
```

si può ricavare il nome dell'URI invocato originariamente prima della forward.

I nuovi attributi aggiunti ricordano molto da vicino quelli introdotti con la versione 2.2 della API

```
javax.servlet.include.request_uri
javax.servlet.include.context_path
javax.servlet.include.servlet_path
javax.servlet.include.path_info
javax.servlet.include.query_string
```

i quali però offrono informazioni relativamente ad operazioni di include e non di forward.

Un'altra importante innovazione è quella che vede la possibilità di specificare il comportamento dei listener in concomitanza delle varie tipologie di operazioni eseguite: questo era un aspetto lasciato in sospeso in precedenza, tanto che il comportamento finale dell'applicazione era non sempre lo stesso al variare del container.

È stato introdotto un nuovo tag <dispatcher> nel deployment descriptor che accetta i seguenti valori: REQUEST, FORWARD, INCLUDE ed ERROR.

In questo modo un filtro può essere attivato o disattivo a piacimento semplicemente editando il file web.xml. Ad esempio

```
<filter-mapping>
  <filter-name>Logging Filter</filter-name>
  <url-pattern>/find/*</url-pattern>
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

in questo caso si vuole specificare che il filtro venga eseguito sia in occasione di request del client, che di operazioni di forward. I valori INCLUDE ed ERROR permettono rispettivamente di attivare il filtro sulle operazioni di include e per le richieste verso la pagina di errore (tag <error-page>).

Se non viene specificato nessun elemento di tipo <dispatcher> verrà utilizzato il valore di default corrispondente a REQUEST.

Ascoltatori: novità fra i ServletListeners

Con la 2.4 è stato esteso il funzionamento dei listener, introdotti con la 2.3 i quali potevano ascoltare solamente eventi legati al contesto (creazione o rimozione oppure aggiunta o rimozione di attributi al contesto stesso).

Adesso è possibile creare ascoltatori che ricevano notifica anche quando viene creata o distrutta una request, oppure quando sono aggiunti o rimossi attributi alla request.

La API 2.4 aggiunge le seguenti classi

```
ServletRequestListener
ServletRequestEvent
ServletRequestAttributeListener
ServletRequestAttributeEvent
```

le quali vanno ad aggiungersi alle abituali ServletContextListener, ServletContextEvent, ServletContextAttributeListener e ServletContextAttributeEvent: la configurazione di questi oggetti è realizzata al solito tramite opportuni tag XML da inserire nel file web.xml.

La nuova specifica chiarisce finalmente un aspetto lasciato in sospeso fino ad oggi, ovvero cosa debba succedere quando un listener lancia una eccezione. Dato che i listener di fatto vivono al di fuori della *service call stack*, l'eccezione non può essere propagata alla servlet (e quindi, nel caso, notificata al client).

L'eventuale errore verrà gestito tramite la direttiva <error-page> se definita o, in alternativa, da un semplice messaggio verso il client corrispondente al codice di errore HTTP 500.

Sessione

Una delle innovazioni più importanti nella nuova API risiede nel metodo HttpSession.logout(), che di fatto serve per invalidare una sessione e quindi per implementare in una applicazione web il meccanismo di logout nel caso in cui l'utente si sia collegato tramite uno dei meccanismi standard di autenticazione (BASIC, DIGEST, FORM, CLIENT-CERT).

Purtroppo il metodo, per quanto utile, non funziona in tutti quei casi in cui le credenziali di autorizzazione sono mantenute dal client (di fatto nei metodi BASIC, DIGEST, e CLIENT-CERT).

Anche se si invalida la sessione, niente impedisce al browser di inviare nuovamente i dati di login, e di fatto questo implica il re-login automatico.

Questa è anche una spiegazione indiretta del perché la maggior parte delle applicazioni web non implementano meccanismi di login tramite FORM, i quali possono essere facilmente invalidati annullando la sessione di lavoro, o annullando i cookie lato client.

Altra modifica relativa alla gestione delle sessioni è quella che prevede la possibilità di definire una valore nullo o negativo come tempo di scadenza della sessione stessa (tag <session-timeout>). In questo caso la sessione non dovrebbe scadere mai, cosa di per sé piuttosto rischiosa, ma utile in certi casi particolari.

Infine, nel caso in cui si utilizzino sessioni distribuite, piazzando in sessione un oggetto che non sia serializzato, verrà generata una IllegalArgumentException, evento sicuramente utile al posto di un silente errore applicativo non meglio specificato.

Deprecato l'STM

In Servlet 2.4 è stato reso deprecated il modello SingleThreadModel (STM) il cui significato era quello di rendere thread-safe e quindi sincrona, l'invocazione delle servlet da parte dei molti client HTTP.

Il motivo di questa scelta risiede nell'impossibilità di garantire in modo certo e sicuro la concorrenza delle chiamate, ma anzi fornisce al programmatore un falso senso di sicurezza. L'argomento sarebbe in realtà piuttosto vasto: ne è data una spiegazione abbastanza chiara e completa in [STM].

Schema

Infine l'ultima modifica che è stata introdotta è relativa alla validazione del file XML di deploy. Adesso tale file è definito tramite un XML Schema anche se per compatibilità con le versioni precedenti sono ancora accettati file di deploy basati su DTD.

Anche se il concetto di schema esula dagli scopi di questo capitolo, si ricorderà brevemente che uno schema è un metodo alternativo, basato a sua volta su XML, per definire la struttura di un file XML.

Con la nuova definizione sono state aggiunte nuove imposizioni (l'elemento <role-name> deve essere unico), mentre l'ordine degli elementi all'interno di <web-app> può essere variabile. Il tag <distributable> può apparire un numero arbitrario di volte, mentre <description> adesso supporta un attributo xml:lang ad indicare il linguaggio utilizzato.

Da notare che un servlet container non necessariamente deve validare il file di deploy tramite lo schema, mentre i container J2EE devono farlo.

Una aggiunta ulteriore che è stata fatta al file di deploy è quella relativa al welcome-file: adesso infatti è possibile specificare una servlet url-pattern come file di welcome all'interno di una applicazione.

Considerazioni sulla API 2.4

Al momento non è ancora possibile utilizzare in produzione la nuova specifica 2.4, dato che la maggior parte dei container non supportano le nuove specifiche, che peraltro non sono ancora disponibili in fase definitiva. Per chi fosse interessato a fare qualche prova, si può utilizzare la versione 5.0 (in beta version, vedi [TOM5]) di Tomcat, che supporta già queste nuove funzionalità. Ovviamente si tratta di un ambiente di esecuzione non definitivo e quindi da non utilizzare in ambito di produzione. Non vi sono ancora notizie precise circa il rilascio della versione definitiva della API 2.4

Bibliografia

[API]
Sito web ufficiale della servlet API: http://java.sun.com/products/servlet

[JSR154]
SR-000154 JavaTM Servlet 2.4 Specification (Proposed Final Draft 3)
http://jcp.org/aboutJava/communityprocess/first/jsr154/index3.html

[FACT]
LORENZO BETTINI, *Una Fabbrica di oggetti*, "MokaByte" n. 27, febbraio 1999
http://www.mokabyte.it/1999/02/factory_teoria.htm

[STM]
AA VV, *Servlet Best Practices, Part 1*, e seguenti
http://www.onjava.com/pub/a/onjava/excerpt/jebp_3/index1.html

[SEC]
SCOTT OAKS, *Java Security*, 2nd Edition, O'Reilly, 2001
http://www.oreilly.com/catalog/javasec2/

[STRU]
Jakarta Apache Struts Web Application Framework, sottoprogetto di Apache Jakarta Project
http://jakarta.apache.org/struts

[Log4J]
The Log4j project, sottoprogetto di Apache Jakarta Project
http://jakarta.apache.org/log4j/docs/index.html

[MBPAK]
MokaPackages, il framework MVC di MokaByte
http://www.mokabyte.it/mokapackages

[MBMVC]
GIOVANNI PULITI, *MokaShop, il negozio online di MokaByte: realizzare applicazioni J2EE multicanale – I parte*, "Mokabyte" n. 60, febbraio 2002
http://www.mokabyte.it/2002/02/devj2ee_1.htm

[TOM5]
http://jakarta.apache.org/tomcat/index.html

Capitolo 6

JSP e Web User Interface

LAVINIO CERQUETTI – PAOLO AIELLO

Introduzione

Un'applicazione può essere definita come un aggregato di conoscenze e di comportamenti (data e business tier) resi accessibili in maniera coerente e funzionalmente unitaria tramite uno strato di presentazione, il quale presiede e regola l'interazione tra il sistema e gli utenti.

Tale layer intermedio costituisce l'interfaccia utente di un'applicazione e rappresenta l'argomento di questo capitolo. In particolare, vogliamo concentrare la nostra attenzione sulla realizzazione di layer UI nel quadro di applicazioni multi-tier web-based in ambiente J2EE.

In questo contesto il ruolo dell'interfaccia utente assume importanza e ampiezza ancora maggiori per due ordini di motivi.

In primo luogo, la caratteristica precipua dei sistemi multi-tier è data dalla distribuzione in componenti discreti e indipendenti delle funzionalità applicative del sistema; tale organizzazione porta indubbi vantaggi in materia di espandibilità, scalabilità, solidità e capacità delle applicazioni di reagire alle caratteristiche del sistema di deployment, al prezzo tuttavia di una significativa frammentazione dello strato di business logic, cosa che rende più complessa l'opera di unificazione concettuale svolta dal layer di interfaccia utente;

In secondo luogo, in ambiti web-based vengono tipicamente richieste agli strati UI precise garanzie di portabilità, universalità ed efficienza, la cui complessità conduce all'utilizzo di appositi framework atti a supportare il design e l'implementazione di interfacce utente in grado di adattare il proprio comportamento ai profili computazionali e di banda delle piattaforme di fruizione.

Il concetto stesso di interfaccia utente in applicazioni Java web-based ha subito profondi mutamenti nel corso di questi anni e — complice anche il crescente successo di J2EE come ambiente onnicomprensivo per la realizzazione di applicazioni distribuite all purpose — ha portato a continue ridefinizioni del ruolo e delle finalità dei layer di presentazione in contesto Java Server.

Nonostante il panorama degli strumenti UI Java sia a tutt'oggi caratterizzato da un numero estremamente elevato di prodotti e framework, spesso affini se non funzionalmente identici, è

corretto affermare che il successo dell'ambiente J2EE stia rapidamente portando a una chiarificazione di ruoli e responsabilità anche per quanto concerne lo strato di presentazione, consegnando agli sviluppatori Java un insieme solido, affidabile, coerente e completo di strumenti e tecnologie in questo senso.

L'elemento fondante delle tecnologie Java rivolte all'implementazione di layer di interfaccia utente in applicazioni multi-tier e web-based è costituito da Java Server Pages, framework in procinto di approdare alla versione 2.0, e dalla JSP Standard Tag Library (JSTL).

Nel corso di questo capitolo studieremo quindi l'utilizzo di JSP nei due suoi principali contesti di utilizzo:

- realizzazione di pagine web HTML a contenuto dinamico in maniera sostanzialmente avulsa dal framework J2EE;

- design e implementazione di interfacce utente ad applicazioni distribuite web-based in ambito J2EE.

In questo secondo contesto valuteremo le nuove funzionalità di JSP introdotte a partire dalla versione 1.2, con particolare riferimento alla libreria JSTL. Rivolgeremo quindi la nostra attenzione a Java Server Faces, la Tag Library JSP destinata a rappresentare la piattaforma standard per la costruzione di interfacce utente web-based componentizzate, estensibili, riusabili e in modalità RAD. Infine, analizzati i meccanismi che permettono agli sviluppatori di estendere il framework Java Server Pages attraverso la realizzazione di Tag Library aggiuntive, concluderemo la nostra discussione con una breve disamina delle novità in cantiere per la futura versione 2.0 di JSP.

Pagine web dinamiche

Il ruolo di Java Server Pages in contesti J2EE

Il framework JSP ha dovuto subire, nelle versioni precedenti alla 1.1, una lunga e motivata serie di critiche che ha inizialmente portato diversi sviluppatori a preferirgli sistemi di templating indipendenti e più specificatamente votati all'implementazione di architetture Model – View – Controller.

Le principali critiche mosse a JSP prendevano spunto dal fatto che tale tecnologia non risultasse ben integrata con le linee guida e l'insieme di best practice e design pattern caratterizzanti lo sviluppo di software distribuito e multi-tier, e che la sua struttura portasse alla realizzazione di applicazioni scarsamente manutenibili e inerentemente non scalabili, caratterizzate da un'insufficiente separazione di ruoli logici e responsabilità funzionali tra i diversi software layer.

A queste considerazioni si aggiunga come, in seguito alle evoluzioni strutturali del panorama di API e protocolli Java e alla sempre maggiore disponibilità di soluzioni alternative, il ruolo di JSP nella realizzazione di applicazioni Java Enterprise e le sue effettive possibilità e opportunità di integrazione con strumenti di valore già noto e assodato siano risultati a lungo poco chiari e abbiano costituito una continua fonte di dubbio per la stessa comunità degli sviluppatori.

Risposte concrete a queste critiche non si sono tuttavia fatte attendere. Un primo risultato è stato raggiunto, nel 2001, con l'introduzione del JSP Model 2, approccio strutturale volto a rendere compatibile il framework JSP con architetture Model – View – Controller e con i pattern fondamentali utilizzati nello sviluppo di strati di presentazione per software distribuito (Business Delegate, Composite View, Service to Worker e View/Helper).

Tale chiarimento "filosofico" del ruolo di JSP è stato accompagnato, a partire dalla versione 1.1 delle specifiche, da innovazioni tecniche che hanno costituito il presupposto per l'evoluzione presente e futura di questa tecnologia, da un lato nell'ottica di un generale miglioramento della sua integrabilità e user-friendliness, dall'altro nel senso di un suo effettivo inquadramento all'interno dell'architettura J2EE come software layer deputato alla gestione del presentation tier a servizi remoti e componenti distribuiti in client web-based.

Delle nuove possibilità offerte da JSP la libreria JSTL costituisce al contempo un esempio e un'applicazione fondamentale: tale Custom Tag Library diverrà ufficialmente un componente standard con l'imminente specifica 2.0 di JSP e costituisce il tanto atteso punto di svolta nell'evoluzione di questo framework, al punto che non è errato asserire che esistano due JSP: un JSP ante-JSTL e un JSP post-JSTL.

JSTL: JSP Standard Tag Library

JSP può essere essenzialmente descritto come un ambiente per la produzione di contenuti web dinamici che si materializzano grazie alla possibilità di innestare, all'interno di componenti statici di presentazione — tipicamente implementati tramite pagine HTML — comportamenti aggiuntivi i cui esiti vengono risolti solamente a run-time.

Tali comportamenti, abbiamo visto, venivano espressi nelle prime versioni di JSP attraverso dei brani di codice Java inseriti direttamente nelle pagine HTML e denominati scriptlet, ovvero tramite dei tag proprietari (*action* in gergo JSP) normalmente associati al namespace `jsp` e usualmente utilizzati per accedere a proprietà e metodi di oggetti esterni.

Purtroppo tale utilizzo di JSP, oltre ad essere caratterizzato da una quantità di problematiche irrisolte a livello di design, portava a delle pagine HTML estremamente difficili da gestire e manutenere, rendendo nella pratica difficilmente realizzabile quella separazione di ruoli tra sviluppatori software e grafici web che è alla base delle moderne pratiche collaborative di software development in architetture complesse e multi-tier.

Il framework JavaServer Pages, a partire dalla versione 1.1, ha introdotto la possibilità per gli sviluppatori di arricchire il contesto di esecuzione delle pagine JSP di nuovi comportamenti, espressi tramite tag aggiuntivi e redatti sotto forma di classi Java raggruppate in contenitori denominati Custom Tag Library, ovvero JSP Tag Extensions; il passo è stato breve, poi, da qui alla formalizzazione di una libreria di action standard, con la quale fornire ai programmatori un patrimonio comune di tag in grado da un lato di garantire la sostanziale compatibilità delle pagine JSP al variare del web container, e dall'altro di fornire quei costrutti di programmazione necessari per svincolare il framework di Sun dal continuo ricorso agli scriptlet Java.

Tale libreria di tag, sviluppata in seno al Java Community Process dall'Apache Group, è appunto JSTL. Tecnicamente parlando, essa consiste in una collezione di Custom Tag Library, organizzate e suddivise per categorie funzionali.

Tabella 6.1 — *Le quattro Custom Tag Library di JSTL con relativi URI e prefix.*

Libreria	URI	Namespace
core	http://java.sun.com/jstl/core	c
XML processing	http://java.sun.com/jstl/xml	x
I18N capable formatting	http://java.sun.com/jstl/fmt	fmt
relational db access (SQL)	http://java.sun.com/jstl/sql	sql

Si noti che JSTL utilizza funzionalità e API delle Custom Tag Library introdotte a partire dalla versione 1.2 di JSP, e non risulta pertanto compatibile con web container che implementino versioni precedenti di questo framework.

Come già detto, JSTL diverrà un componente standard di JSP — e quindi dei differenti J2EE Server — a partire dalla versione 2.0 di JavaServer Pages; sino ad allora sarà necessario provvedere esplicitamente alla sua installazione, usualmente copiandone la distribuzione nella directory WEB-INF/lib delle singole applicazioni J2EE.

Componenti fondamentali di JSTL

JSTL si compone di quattro Custom Tag Library, ognuna delle quali viene mappata in namespace diversi (detti anche *prefix* in gergo JSP) e associata a URI differenti e stabiliti per convenzione (tab. 6.1).

- *Core*: tale libreria fornisce tag sostanzialmente paralleli ai costrutti strutturati di programmazione in Java, al fine di ridurre al minimo il bisogno di innestare scriptlet all'interno delle pagine JSP. La libreria *Core* implementa inoltre alcune utility action volte alla manipolazione di URL e contenuti esterni.

- *XML processing*: libreria orientata alla gestione di documenti XML e XSLT.

- *I18N capable formatting*: libreria di tag rivolta alla localizzazione delle applicazioni.

- *relational db access (SQL)*: libreria di tag rivolta all'utilizzo di connessioni JDBC; si noti che tale libreria, in maniera piuttosto evidente, non rispetta la separazione dei ruoli tipici delle applicazioni multi-tier, dal momento che consente di integrare sezioni di business logic all'interno di pagine JSP, vale a dire del presentation layer. Tuttavia la disponibilità di facility per la gestione diretta di database internamente allo strato di interfaccia utente gioca un ruolo importante nell'implementazione di sistemi prototipo e di applicazioni di complessità elementare — oltre ad avere un preciso appeal a livello di marketing — ragion per cui questa libreria è stata inclusa come componente standard nella versione 1.0 di JSTL.

Ogni libreria viene dichiarata nelle pagine JSP che ne richiedono l'utilizzo tramite la sintassi:

```
<%@ taglib prefix="<namespace>" uri="<URI>" %>
```

Ad esempio la dichiarazione per la libreria *Core* leggerà:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
```

Per quanto gli URI e i prefissi associati alle singole librerie siano da intendersi solamente come raccomandazioni formali, è tuttavia buona pratica attenersi agli standard indicati per due ordini di motivi:

- seguendo tali standard si contribuisce a incrementare la comprensibilità e la riusabilità del codice;
- l'utilizzo degli URI segnalati permette ai singoli J2EE Server di "riconoscere" le librerie utilizzate e di implementare eventuali politiche di ottimizzazione.

Gli Expression Language

Perché una pagina JSP risulti di una qualche utilità è indispensabile che essa sia in grado di rapportarsi alla ricca Object API che contraddistingue il mondo Java; sorge in altre parole il problema di come sia possibile accedere a oggetti esterni e combinarne le proprietà in espressioni semanticamente significative. A questo problema sono state date, nel corso dell'evoluzione di JSTL, due risposte differenti:

- EL: tale acronimo sta semplicemente per *Expression Language*, un linguaggio estremamente immediato e user-friendly per l'accesso a oggetti e per l'utilizzo delle loro proprietà a runtime dall'interno di pagine JSP. EL, per la sua sintassi semplificata — volutamente ispirata a ECMAScript e XPath — e in virtù del fatto che non richiede conoscenze specifiche in area Java, rappresenta lo strumento consigliato nell'utilizzo di JSTL, ed è l'Expression Language su cui ci soffermeremo nel corso di questo capitolo.
- RT (*Request-Time* expression language): questo linguaggio, ormai obsoleto, consente di utilizzare all'interno di tag JSTL espressioni specificate direttamente nel linguaggio di scripting della pagina, vale a dire usualmente in linguaggio Java, in maniera assai simile a quanto permesso dall'ormai famigerato tag `<%=`.

Espressioni scritte in un Expression Language possono essere utilizzate sia negli attributi di tutti i tag JSTL — con l'eccezione dei soli attributi `var` e `scope` — sia negli attributi di eventuali Custom Tag Library di terze parti o proprietarie.

Il linguaggio EL

Come anticipato, la sintassi del linguaggio EL è sorprendentemente semplice: ogni espressione EL è racchiusa all'interno delle sequenze di caratteri `${` e `}`, ed è possibile accedere alle proprietà di un oggetto sia secondo la tradizionale sintassi basata sull'operatore " . " (punto) sia consideran-

do l'oggetto alla stregua di un vettore associativo, le cui proprietà costituiscono le chiavi del vettore. Ad esempio, per accedere alla proprietà id dell'oggetto customer (implementata dai metodi getId() e setId()) è possibile utilizzare le seguenti sintassi:

```
${customer.id}
${customer["id"]}
```

Il vantaggio della seconda sintassi — utilizzabile anche per estrarre valori da oggetti che implementano l'interfaccia java.util.Map — risiede, in maniera abbastanza ovvia, nella possibilità di decidere a runtime la proprietà cui si desidera accedere, memorizzandone il nome in una variabile a cui si accede tramite una sintassi del tipo:

```
${customer[propertyName]}
```

L'immediatezza di EL si manifesta anche nella sintassi per l'accesso a oggetti di tipo vettore, vale a dire array Java ovvero oggetti che implementino l'interfaccia java.util.List:

```
${customerList[0]}
```

EL supporta gli usuali operatori logici e aritmetici, per alcuni dei quali sono definiti dei sinonimi al fine di mantenere la compatibilità sia con ECMAScript che con XPath e per evitare il bisogno di ricorrere a entity qualora si utilizzi JSP in modalità XML.

Ogni oggetto definito in uno scope JSP valido viene automaticamente riconosciuto e può essere utilizzato liberamente all'interno delle espressioni EL; gli scope JSP sono esaminati alla ricerca degli oggetti specificati dall'utente, nel seguente ordine:

- scope di pagina;
- scope di richiesta HTTP;
- scope di sessione;
- scope di applicazione.

EL definisce inoltre una serie di oggetti cosiddetti impliciti, tramite i quali risulta possibile accedere al contesto corrente di pagina, ai valori degli attributi con scope di richiesta HTTP, di pagina e di sessione, nonché a cookie, request header e parametri di inizializzazione. EL riconosce, oltre a oggetti e variabili, costanti espresse nei formati consueti:

- Costanti stringa: sequenze di caratteri racchiuse tra apici singoli o doppi, al cui interno il carattere di backslash viene utilizzato per identificare dei caratteri di controllo convenzionali ('\n', '\t', etc.); per inserire un carattere di backslash all'interno di una costante stringa, lo stesso deve essere preceduto da un altro backslash.

- Costanti intere: sequenze numeriche precedute da specificatore opzionale di segno (+ o -).
- Costanti decimali: sequenze numeriche precedute da uno specificatore opzionale di segno e composte di una parte intera e di una parte decimale, separate tra di loro dal punto decimale " . "; la parte decimale può essere espressa in notazione esponenziale.
- Costanti booleane: true e false.

La libreria JSTL core

Laddove EL definisce un linguaggio per la specifica di espressioni, la libreria JSTL *Core* definisce i tag che implementano i controlli di flusso e i costrutti strutturati grazie ai quali è possibile esprimere algoritmi, all'interno di pagine JSP, senza più dover ricorrere al tag di escape in Java <%.

Nello spirito di JSTL, anche i tag sono caratterizzati da una notevole immediatezza d'uso e facilità di comprensione, al punto da risultare facilmente intelligibili anche senza conoscerne in dettaglio le caratteristiche. A titolo di esempio la visualizzazione dei nominativi di tutti i contatti memorizzati in un ipotetico address book potrebbe essere realizzata con un codice simile al seguente (il namespace c si riferisce, come da convenzione, alla libreria JSTL Core):

```
<table border="1">
<tr><td>Contatto</td></tr>
<c:forEach var="contact" items="${addressBook.contacts}">
  <tr><td><c:out value="${contact.name}" /></td></tr>
</c:forEach>
</table>
```

Tale codice, utilizzando le tradizionali pratiche di programmazione JSP basate su scriptlet Java, avrebbe un aspetto simile al seguente:

```
<jsp:useBean id="addressBook" scope="..." />
<table border="1">
<tr><td>Contatto</td></tr>
<%
  Iterator i=addressBook.getContacts().iterator();
  while (i.hasNext()) {
    Contact contact=(Contact)i.next();
%>
  <tr><td><%= contact.getName()%></td></tr>
<%
    }
%>
</table>
```

I vantaggi in termini di leggibilità e manutenibilità del codice sono immediatamente evidenti. Si noti anche come l'utilizzo di EL e della sua funzionalità automatica di ricerca degli oggetti nella catena degli scope correnti ci dispensi dall'utilizzo dei tag <jsp:useBean>.

I tag della libreria JSTL Core, da un punto di vista logico, rientrano in tre macrocategorie.

1. Tag di supporto al linguaggio di espressioni: il fine di questi tag è quello di fornire un "ponte" tra il linguaggio di espressioni utilizzato e la libreria JSTL Core. Appartengono a questo gruppo i seguenti tag:

   - out: utilizzato per valutare un'espressione — specificata nell'attributo value — e inviarne il risultato al JspWriter corrente, in maniera analoga a quanto accade con il tag <%=;

   - set: imposta il valore di un attributo JSP o di una proprietà di un oggetto;

   - remote: elimina un attributo/oggetto dallo scope specificato;

   - catch: permette di gestire errori non critici direttamente dall'interno della pagina JSP.

2. Tag di controllo di flusso: il fine di questi tag è di fornire versioni JSTL dei costrutti di iterazione e di scelta:

   - if: implementa un costrutto di scelta analogo allo statement if del linguaggio Java;

   - choose, when e otherwise: questi tag implementano un costrutto di scelta multipla in maniera simile agli statement Java switch, case e default;

   - forEach: realizza un costrutto di looping, sia basato su iteratore, come nell'esempio visto in precedenza, sia — in base agli attributi forniti al tag — in forma più generica, analogamente allo statement for di Java; si noti che forEach è in grado di iterare automaticamente su array di tipi primitivi, Collection, Iterator ed Enumeration;

   - forTokens: implementa un costrutto di iterazione tramite parsing di una stringa in maniera simile a quanto realizzato dalla classe StringTokenizer.

3. Tag di gestione di URL:

   - import: utilizzato per rendere disponibile sotto forma di variabile il contenuto di un URL; questo tag può essere utilizzato semplicemente per includere il contenuto di un URL esterno all'applicazione nella pagina JSP corrente — azione non supportata dal tag <jsp:include> — ovvero per importare ed elaborare dati remoti, magari in formato XML, con l'ausilio della libreria JSTL XML processing;

   - url: permette la costruzione di URL a runtime, in base agli attributi specificati e al contesto dell'applicazione corrente;

- redirect: provoca una browser redirection.

Un esempio di utilizzo

Intendiamo ora esemplificare i vantaggi di JSTL attraverso il design e la realizzazione di una semplice applicazione multi-tier, la quale ci seguirà nel corso di tutto il capitolo e che utilizzeremo come banco di prova delle tecnologie con cui andremo a fare via via conoscenza. Tale applicazione, denominata NetView e liberamente disponibile in allegato al libro, sarà rivolta al mantenimento e al controllo di un semplice sistema informativo d'impresa.

Dal momento che, per ovvi motivi di spazio, i sorgenti non possono venire integralmente presentati in questa sede, si consiglia il lettore di procedere all'installazione di NetView e di tenerne il codice sempre a portata di mano durante la lettura del presente capitolo, dal momento che esso costituisce un ausilio fondamentale per la comprensione degli argomenti trattati.

La nostra web application si interfaccerà sia ai diversi database della ditta committente, contenenti l'inventario aggiornato dei sistemi da monitorare e i loro profili di configurazione, sia a ipotetici sistemi real-time di verifica del carico; i dati provenienti da entrambe queste sorgenti verranno integrati trasparentemente così da creare l'illusione di un singolo repository centrale. Si richiede inoltre che l'applicazione sia portabile e indipendente da piattaforme specifiche e da eventuali moduli software installati presso i sistemi degli utenti.

Analisi del problema

La struttura inerentemente distribuita dell'applicazione si presta assai bene a una sua realizzazione in un contesto J2EE e le caratteristiche dell'interfaccia e i requisiti di universalità dell'accesso ci indirizzano in maniera naturale all'implementazione di un client web-based. Come ogni applicazione multi-tier che si rispetti, anche in NetView possiamo individuare più blocchi logici separati, che andremo ora ad analizzare.

Il sistema informativo

Un'implementazione completa del sistema informativo è, ovviamente, al di là dei nostri interessi; per i nostri fini viene fornita una semplice implementazione costituita da una classe singleton InformationSystem, che rappresenta la visione unitaria del mondo back end di NetView e implementa il pattern Business Delegate.

Il nostro sistema informativo è basato su due tabelle: la tabella Nodi, che contiene sostanzialmente l'inventario dei dispositivi che dobbiamo monitorare, e la tabella Categorie, che contiene l'elenco delle classi di dispositivi gestibili (Router, Application Server, etc.). Tali tabelle possiedono chiavi primarie elementari e sono collegate tra di loro tramite una relazione uno a molti (Categorie —> Nodi).

Lo strato di business logic

Nell'implementazione d'esempio lo strato di business logic è fuso all'interno della classe InformationSystem, e esporta all'interfaccia utente i contenuti del "database" attraverso oggetti facenti capo ad apposite classi lightweight.

La versione proposta di NetView conosce due classi di questo genere, CategoryRow e NodeRow, i cui oggetti corrispondono rispettivamente alle righe delle tabelle Categorie e Nodi.

Alla classe NodeRow spetta anche il compito di integrare informazioni non ottenute dal nostro ipotetico database: a questo fine, per simulare una lettura da sistemi remoti di controllo, le colonne Carico attuale e Data e ora di ultima rilevazione del carico della tabella Nodi vengono rigenerate a ogni accesso.

Lo strato di interfaccia utente

A livello di presentation tier si pone il problema della separazione tra forma e contenuti, la cui soluzione ideale consiste in un disaccoppiamento formale e rigoroso.

La struttura implementata rispetta questi requisiti e separa chiaramente le funzionalità di presentazione dei dati da quelle della loro preventiva elaborazione e della comunicazione con gli strati superiori del sistema, ed è basata su un'architettura integrale MVC e sui pattern Front Controller e Service to Worker.

Un nodo fondamentale di ogni applicazione distribuita, infine, è costituito dai servizi di autenticazione/autorizzazione e tracing; nel contesto del nostro esempio, tuttavia, tali aree funzionali non rivestono per noi alcun interesse e, non essendo legate alla comprensione dei meccanismi e delle dinamiche di funzionamento e integrazione di JSP e JSTL, non verranno prese in considerazione dall'implementazione di NetView.

La struttura generale dell'applicazione

La struttura di NetView è basata sul pattern FrontController, implementato tramite una servlet la cui responsabilità consiste unicamente nel ricevere dal browser la richiesta di una determinata azione da parte dell'utente e nel procedere al forwarding di tale richiesta a un'appropriata classe di gestione Java, genericamente indicata come dispatcher. Tale classe restituisce l'URL di un componente dinamico — denominato View e nel nostro caso implementato per mezzo di pagine JSP — il quale implementa la vera e propria interfaccia utente e che verrà attivato dal Front Controller.

Analisi dell'interfaccia utente

Dal momento che la nostra attenzione verte essenzialmente sulla struttura e sulle funzionalità di UI, ha senso analizzare NetView a partire dalle principali azioni cui l'applicativo è in grado di rispondere.

La prima azione che considereremo è front: essa fa capo alla view front.jsp, la cui prima riga — come per tutte le view — dichiara l'utilizzo della libreria JSTL Core:

```
"<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>".
```

Il compito dell'azione front è presentare la home page della nostra applicazione, la quale visualizza un messaggio di benvenuto unitamente alla data e ora attuali e alla stringa di identificazione del browser dell'utente.

La prima informazione viene ottenuta mediante una semplice JSP action <jsp:useBean> tramite la quale viene istanziato un oggetto java.util.Date, rigenerato a ogni accesso alla pagina e visualizzato

tramite un tag JSTL <c:out>. Come ci si può attendere, la visualizzazione di un oggetto Java passa per la sua rappresentazione sotto forma di stringa, ottenuta automaticamente da JSTL invocando il relativo metodo toString().

L'identificazione del browser è ancora più semplice, e consiste semplicemente nella visualizzazione della proprietà user-agent dell'oggetto header corrispondente al request header della richiesta HTTP corrente. Trattandosi di un oggetto implicito non è ovviamente richiesta la sua dichiarazione tramite <jsp:useBean>.

La home page — come ogni pagina di NetView — include le due risorse esterne topbar.jsp e bottombar.jsp.

La struttura delle pagine di NetView è in effetti composta di tre sezioni: una barra superiore, un frame di contenuti e una bottom bar. Questa struttura, con diverse variazioni, è una costante di gran parte delle web application. Le due barre hanno il compito di realizzare un look & feel standard che dia un senso di continuità tra le diverse pagine prodotte dall'applicazione.

Il frame di contenuti rappresenta la parte dell'interfaccia che effettivamente varia durante la navigazione all'interno dell'applicazione. Nella nostra implementazione ogni view (ossia ogni pagina JSP) si preoccupa unicamente del rendering di tale frame, e si limita a utilizzare le due risorse suddette per le altre sezioni.

La barra superiore ha inoltre la funzione di menu principale, e consente all'utente di accedere alle funzionalità di visualizzazione dei nodi, sia globale che per categoria. La selezione della categoria di visualizzazione dei nodi avviene tramite un'interazione JavaScript/JSTL così realizzata:

```
<form method="post" name="nodesbyidForm" action="/netview/main">
  <input type="hidden" name="action" value="nodesbycatid" />
  <select name="catId" onChange="document.nodesbyidForm.submit();" >
    <c:forEach var="category" items="${informationSystem.allCategories}">
      <option value="<c:out value='${category.id}'/>">
        <c:out value="${category.desc}" />
      </option>
    </c:forEach>
    </select>
</form>
```

La comprensione di questo codice è la chiave per l'utilizzo corretto di JSTL: sorgenti di NetView alla mano, i punti salienti di queste righe sono quelli riportati di seguito.

Tag <select>: apriamo una listbox, che in seguito alla selezione di un elemento causerà automaticamente il post del form, portando all'esecuzione dell'azione nodesbycatid (visualizza nodi per categoria).

Tag <c:forEach>: iteriamo sulla proprietà allCategories di informationSystem, un riferimento al nostro sistema informativo che viene reso disponibile ad ogni view; il suo metodo getAllCategories() ritorna l'elenco di tutte le categorie disponibili sotto forma di Collection di oggetti CategoryRow; la variabile category, dichiarata nell'attributo var del tag <c:forEach>, sarà il nostro iterator su tale Collection.

A questo punto per ogni elemento della suddetta Collection, vale a dire per ogni CategoryRow, verrà eseguito il codice sino alla chiusura del tag <c:forEach>.

Per ogni CategoryRow viene generato un tag <option>, il cui attributo value è pari al valore visualizzato dal tag <c:out value='${category.id}' />, vale a dire al codice della categoria (metodo getId() di CategoryRow). L'innesto di tag JSTL e HTML è perfettamento legittimo e completamente trasparente al client: visualizzando il sorgente HTML all'interno del browser si noterà che i tag JSTL sono stati sostituiti dal loro valore. Nei sorgenti riportati, per chiarezza, in ogni innesto JSTL/HTML si è provveduto a utilizzare caratteri terminatori di stringa differenti (" e ') anche se ciò non è assolutamente richiesto da JSP.

Il valore visualizzato in ogni <option> all'interno della list box è ${category.desc}, vale a dire la descrizione della categoria (metodo getDesc() di CategoryRow).

Si noti che né l'oggetto informationSystem, né l'oggetto category vengono dichiarati o definiti in azioni JSP <jsp:useBean>; nel primo caso, infatti, ci limitiamo a utilizzare il meccanismo di ricerca automatica di JSTL che raggiunge automaticamente gli oggetti nei contesti di pagina, richiesta HTTP, sessione e applicazione. Nel secondo caso è invece sufficiente la definizione di tale elemento come Iterator nell'attributo var del tag <c:forEach>.

La top bar e la bottom bar, il cui contenuto è tendenzialmente statico, vengono incluse da ogni view con delle direttive <%@ include %>.

Selezionando Tutti i nodi dal menu principale, viene eseguita l'azione allnodes: tale azione, attraverso il dispatcher AllNodesDispatcher, porta all'attivazione della view nodes.jsp. Selezionando invece dal menu principale una categoria, viene eseguita l'azione nodesbycatid, la quale attraverso il dispatcher NodesByCatIdDispatcher ci conduce alla medesima vista nodes.jsp.

Tale view è in grado di riconoscere il proprio contesto di invocazione in base al valore del parametro action, che viene reso disponibile ad ogni view.

In ambedue i casi la nostra view creerà una tabella HTML per visualizzare i dati della collection nodes, che viene inserita nel contesto della richiesta di pagina dai dispatcher. Il dispatcher AllNodesDispatcher inserisce in tale attributo l'elenco di tutti i nodi disponibili mentre il dispatcher NodesByCatIdDispatcher vi inserisce i soli nodi appartenenti alla categoria richiesta.

Nel caso di azione nodesbycatid viene anche passato alla view un oggetto CategoryRow corrispondente alla categoria richiesta, in modo che la pagina JSP possa riproporre all'utente il criterio da lui impostato.

Consideriamo ora brevemente come la view nodes.jsp utilizzi il valore del parametro action per impostare correttamente il titolo HTML della pagina:

```
<title>NetView -
  <c:choose>
    <c:when test="${action=='allnodes'}">Tutti i nodi</c:when>
    <c:when test="${action=='nodesbycatid'}">Nodi per categoria</c:when>
  </c:choose>
</title>
```

La leggibilità di JSTL è sicuramente uno dei suoi maggiori pregi, anche a costo di una certa ridondanza.

L'esperienza nell'utilizzo di questa tag library insegna che, proprio per via di tale prolissità, l'espressione di algoritmi non elementari in JSTL è estremamente scomoda. Questa caratteristica è paradossalmente da intendersi come assolutamente positiva, dal momento che conduce in maniera naturale all'estrapolazione di tali algoritmi in oggetti Java esterni di tipo View Helper, e quindi all'implementazione di soluzioni distribuite corrette, flessibili e manutenibili.

La produzione dell'elenco dei nodi in forma tabellare è estremamente semplice, e consiste nell'iterazione sulla collezione di oggetti NodeRow e nella visualizzazione delle proprietà di ogni nodo all'interno di corrispondenti elementi <td>.

Esaminiamo ora l'azione nodeform: essa viene invocata dalla view nodes.jsp per visualizzare i dati di un singolo nodo all'interno di un form per permetterne l'edit (subaction edit con parametro id pari al codice del nodo) ovvero per effettuare l'inserimento di un nuovo nodo (subaction add).

Tale azione porta all'esecuzione del dispatcher NodeFormDispatcher, il quale ha il compito di rendere disponibili alla view i dati da precaricare nel form, vale a dire — in caso di modifica — i dati del nodo selezionato, e in caso di inserimento il progressivo automatico del codice del nodo.

Per far questo, NodeFormDispatcher utilizza il parametro id per risalire ai valori del nodo, che inserisce uno a uno nel contesto di pagina.

Tali attributi verranno proposti come valori di default nel form di edit dalla view nodeform.jsp, ancora una volta innestando JSTL in HTML:

```
<input type="text" name="desc" value="<c:out value='${desc}' />" size="50" />
```

Tale view utilizza il meccanismo già visto in topbar.jsp per produrre la listbox delle categorie, questa volta arricchito di un <c:if> per selezionare automaticamente l'elemento della listbox corrispondente alla categoria del nodo corrente.

Un <c:if> viene utilizzato anche per tradurre lo stato del nodo (un carattere tra C, T e P) in un radio button di facile utilizzo.

Si noti che alla tecnica di innesto di JSTL / HTML si affiancherà, nelle specifiche di JSP 2.0, una soluzione alternativa chiamata *implicit scripting*, la quale permette di utilizzare espressioni in linguaggio EL liberamente all'interno delle pagine web. La linea HTML precedente potrebbe quindi essere riscritta così:

```
<input type="text" name="desc" value="${desc}" size="50" />
```

I vantaggi dell'implicit scripting, in termini di leggibilità, sono piuttosto ovvi, ma vengono bilanciati da possibili rischi di incompatibilità con pagine HTML che già utilizzano le sequenze di caratteri ${ e }.

Tale tecnologia è in effetti già disponibile in alcuni Application Server — quali Tomcat 5 — ma non è stata da noi utilizzata nell'implementazione di NetView per garantire una maggiore compatibilità nei confronti dei diversi ambienti J2EE Server attualmente disponibili.

In seguito alla conferma del form viene eseguita l'azione nodepost. A tale azione, cui si può arrivare anche direttamente dalla view nodes.jsp confermando la cancellazione di un nodo, spetta la responsabilità di riportare nella base dati l'operazione di inserimento, modifica o cancellazione di nodi effettuata dall'utente.

Il dispatcher NodePostDispatcher recupera in caso di inserimento o modifica tutti i dati del nodo inseriti dall'utente del form e li utilizza per riempire una nuova istanza di NodeRow, di cui poi richiede la persistenza a InformationSystem.

L'esito della richiesta di persistenza viene tradotto in un assegnamento all'attributo di richiesta HTTP result, che viene utilizzato dalla view nodepost.jsp per fornire all'utente un opportuno messaggio di conferma o di errore.

Componenti aggiuntivi di JSTL: XML, SQL e internazionalizzazione

Alla libreria Core di JSTL si affiancano tre altre tag library, rivolte alla gestione via JSTL di documenti XML, all'accesso a database in modalità SQL e all'implementazione di applicazioni web internazionalizzate. Sperimenteremo sul campo le funzionalità di tali componenti di JSTL assumendo che il committente di NetView abbia richiesto l'implementazione di alcune nuove funzionalità:

- tutti i dispositivi, tranne i router, inviano dei log operativi a un repository centrale; gli operatori che utilizzano NetView devono essere messi in grado di analizzare e filtrare tali log, memorizzati in formato XML;
- il committente possiede già un sistema di monitoraggio e controllo del traffico, il quale preleva i relativi dati dai router aziendali e li consolida in totali giornalieri, archiviandoli in un database relazionale; si richiede a NetView di interfacciarsi a tale database — ovviamente in sola lettura — in maniera da permettere agli operatori di effettuare analisi elementari del traffico senza essere costretti a lasciare l'applicazione;
- nell'ottica di un progetto pilota incentrato su un possibile ampliamento di responsabilità di NetView, si richiede l'internazionalizzazione della nostra piccola applicazione, la quale deve essere messa in grado di adattarsi automaticamente alla lingua e alle convenzioni di rappresentazione di numeri e date di Paesi diversi.

Dal momento che, almeno in una prima fase, tali funzionalità aggiuntive vanno considerate sperimentali, decidiamo di mantenere l'architettura di NetView immutata: il nostro fine è quello di implementare tutte le richieste lavorando quasi esclusivamente a livello di interfaccia utente, ovvero di view JSP, avvalendoci della ricchezza semantica di JSP e JSTL al fine di minimizzare gli interventi necessari sul codice Java.

Faremo quindi uso largo uso dei componenti aggiuntivi di JSTL, e in particolare:

- intendiamo realizzare le funzionalità di accesso ai log operativi tramite il componente XML della libreria JSTL;
- l'implementazione delle funzioni di report sarà interamente delegata alle view JSP, al cui interno utilizzeremo componenti per l'accesso a DB relazionali della JSP Standard Tag Library;

- verremo incontro ai desideri del cliente in tema di internazionalizzazione, ancora una volta, operando unicamente nel presentation layer e avvalendoci dell'apposito componente di JSTL, il cui fine è proprio quello di isolare l'interfaccia utente dalle specificità geografiche dei contesti di fruizione.

Funzionalità XML di JSTL

Le funzionalità XML di JSTL — nello spirito di questa tag library — sono caratterizzate da estrema semplicità e linearità, e il loro apprendimento viene grandemente semplificato dalla stretta analogia dei tag XML con quelli, a noi già noti, del componente Core di JSTL.

La volontà di utilizzare i componenti XML di JSTL all'interno di una web application va innanzitutto specificata a livello di deployment descriptor:

```
<taglib>
  <taglib-uri>http://java.sun.com/jstl/xml</taglib-uri>
  <taglib-location>/WEB-INF/x.tld</taglib-location>
</taglib>
```

La tag library XML di JSTL va quindi dichiarata in ogni pagina JSP da cui si intenda far uso di tali funzionalità:

```
<%@ taglib prefix="x" uri="http://java.sun.com/jstl/xml" %>
```

Le possibilità di ricerca e estrazione di informazioni da documenti XML vengono implementate nella JSP Standard Tag Library tramite espressioni in linguaggio XPath.

Tale linguaggio, sviluppato e standardizzato in seno al World Wide Web Consortium, offre possibilità di selezione e analisi di informazioni in formato XML estremamente vaste e sofisticate, che noi non utilizzeremo se non in minima parte.

È interessante notare come l'expression language di volta in volta utilizzato sotto JSTL — nei nostri esempi sempre EL, ma quanto detto vale anche per RT — risulti perfettamente integrato con XPath, dall'interno del quale è possibile fare riferimento a variabili e oggetti definiti in contesto JSTL. Espressioni XPath vengono accettate dai tag XML di JSP Standard Tag Library nell'attributo select.

La libreria XML di JSTL non supporta né DTD né XML Schema e non effettua, né in fase di parsing né in fase di applicazione di stylesheet XSLT, alcuna validazione di documenti XML.

La gestione di informazioni XML viene implementata dai seguenti tag:

- <x:parse>: ha il compito di effettuare il parsing di un documento XML, rendendone visibili i contenuti alla pagina JSP; tale documento può essere fornito sotto forma di variabile string — tipicamente valorizzata da una sorgente esterna mediante un precedente tag <c:import> — o sotto forma di reader predisposto dallo strato superiore della nostra applicazione Java; per evitare i problemi di performance legati all'interpretazione di documenti XML complessi è possibile specificare un filtro in formato org.xml.sax.SAXFilter che verrà applicato alla sorgente XML precedentemente alla vera e propria fase di parsing.

- <x:out>: esattamente analogo al tag <c:out>, visualizza il valore di un'espressione specificata in linguaggio XPath nell'attributo select.

- <x:set>: analogo al tag <c:set>, assegna ad una variabile il valore di un'espressione XPath.

- <x:if>, <x:choose>, <x:when>, <x:otherwise>: corrispondono agli omonimi tag del componente Core di JSTL, ma operano su espressioni XPath.

- <x:forEach>: itera su di un'espressione XPath, ritornando uno a uno tutti i nodi di un documento XML selezionati da tale espressione.

- <x:transform>: permette di applicare uno stylesheet XSLT a un documento XML, in maniera eventualmente parametrica e visualizzando ovvero assegnando i risultati a una variabile JSTL.

Implementazione delle funzionalità di analisi dei log

Entriamo nel vivo dell'analisi dei sorgenti partendo dall'implementazione della visualizzazione dei log prodotti dai diversi dispositivi.

Le modifiche al codice Java sono di scarsa rilevanza e consistono unicamente nella definizione di un nuovo dispatcher associato all'azione querylogs e in una modifica alla classe FrontControllerServlet affinché essa inserisca nel contesto di request, prima di invocare il dispatcher, un riferimento all'oggetto ServletConfig corrente.

Il nuovo dispatcher QueryLogsDispatcher, sfruttando tale riferimento, si preoccupa di predisporre un reader verso il file XML contenente i log.

Passiamo ora all'analisi della view querylogs.jsp: la prima sezione del file si occupa semplicemente di visualizzare gli eventuali parametri di ricerca impostati dall'utente e successivamente viene effettuato il parsing del documento XML a partire dalla sorgente dati — nel nostro caso un reader — impostata dal dispatcher QueryLogsDispatcher:

```
<x:parse xml="${logData}" var="parsedLog" scope="request" />
```

L'aver optato per un'interazione basata su reader è dovuto unicamente a motivi didattici; nello spirito della nostra scelta iniziale sarebbe stato perfettamente possibile, e legittimo, rimuovere del tutto il nuovo codice di gestione dal dispatcher e da FrontControllerServlet e leggere direttamente il documento XML dal file logData.xml per poi effettuarne il parsing:

```
<c:import url="WEB-INF/logData.xml" var="logData"
  scope="request" />
<x:parse xml="${logData}" var="parsedLog" scope="request" />
```

Si noti come le variabili definite in contesto Core JSTL risultino trasparentemente utilizzabili all'interno di tag XML JSTL.

Ora che i dati in formato XML sono disponibili in forma normalizzata nella variabile parsedLog è possibile provvedere all'estrazione delle righe di log. La logica consiste semplicemente nell'estrarre tutti i tag <entry> e nel visualizzare solamente quelli che soddisfano i criteri di ricerca

impostati dall'utente. Il senso del codice contenuto in querylogs.jsp risulta immediatamente chiaro grazie alla stretta analogia dei tag XML (<x:forEach>, <x:if>, <x:set> e <x:out>) con i corrispondenti tag Core.

Le espressioni XPath utilizzate all'interno di tale view, alla cui analisi desideriamo rimandare il lettore, hanno il seguente significato:

- "$parsedLog//entry" corrisponde ad ogni tag <entry> all'interno del documento contenuto nella variabile parsedLog, a qualsiasi livello di profondità;

- il valore di "$entry/@producer" è semplicemente il valore dell'attributo producer del tag XML contenuto nella variabile entry; tale variabile, specificata come iteratore nel tag <forEach>, contiene di volta in volta il contenuto del tag <entry> corrente; similmente le espressioni "$entry/@area" e "$entry/@level" valgono rispettivamente il valore dell'attributo area e level;

- il valore delle espressioni "$entry/timestamp" e "$entry/message" corrisponde al contenuto degli elementi <timestamp> e <message> figli del tag contenuto nella variabile entry.

L'espressione XPath "($queryNodeId=-1 or $entry/@producer=$queryNodeId) and ($area='' or $entry/@area=$area) and ($logLevel='' or $entry/@level=$logLevel)" assume il valore true se le seguenti tre condizioni sono soddisfatte:

- l'utente non ha specificato alcun nodo tra i parametri di ricerca (parametro queryNodeId) o il nodo della voce di log corrente corrisponde al nodo richiesto;

- l'utente non ha specificato alcun area di log tra i parametri di ricerca o l'area del nodo di log corrente corrisponde a quella richiesta;

- l'utente non ha specificato alcun livello di log tra i parametri di ricerca o il livello del nodo di log corrente corrisponde a quello richiesto.

Il tag <x:set select="string($entry/@producer)" var="producerId" /> viene utilizzato per assegnare il valore dell'attributo producer alla variabile JSTL producerId; tale valore viene considerato come stringa, vale a dire senza alcuna interpretazione da parte dell'engine XPath. Da questo momento in avanti, la variabile producerId è manipolabile attraverso i normali tag Core, come il tag <c:out> utilizzato subito dopo per inviarne il valore in output.

Funzionalità SQL di JSTL

I componenti SQL di JSTL — come è lecito attendersi dalle nostre precedenti esperienze con questa libreria — sono pochi e chiari, e la loro comprensione risulterà immediata a chiunque abbia utilizzato, almeno una volta nella vita, le JDBC API.

Analogamente a quanto avviene per il componente XML, i servizi SQL di JSTL vanno dichiarati nel deployment descriptor e attivati dall'interno delle singole view JSP con una direttiva <%@ taglib>.

La connessione al database SQL di NetView viene specificata nel deployment descriptor come default per l'intera applicazione tramite un apposito elemento <context-param>.

I componenti SQL permettono di procedere alla configurazione delle connessioni tramite una pluralità di metodi, e precisamente, ogni tag SQL può, nell'attributo dataSource, specificare una connessione attraverso il suo percorso JNDI o tramite la sintassi, da noi utilizzata, avente la forma "jdbc:<dataSource>,<driverClass>,<userName>,<password>"; qualora il tag non possieda attributo dataSource, verrà utilizzata la connessione di default.

In alternativa, ogni tag SQL può, sempre nell'attributo dataSource, fare esplicito riferimento a un oggetto DataSource impostato da uno strato di codice Java superiore o tramite un tag <sql:setDataSource>.

Il tag <sql:setDataSource> rende disponibile un nuovo oggetto DataSource alla pagina corrente, eventualmente impostandolo come nuova connessione di default; tale oggetto può essere ottenuto da uno strato esterno di codice, tramite un riferimento JNDI, tramite la sintassi da noi utilizzata o grazie all'utilizzo diretto degli attributi driver, url, user e password del tag.

Infine, una connessione di default può essere impostata a livello di applicazione direttamente nel deployment descriptor tramite il suo percorso JNDI o la sintassi "jdbc:<dataSource>, <driverClass>,<userName>,<password>" da noi utilizzata.

È da notare come la gestione delle connessioni SQL di JSTL risulti perfettamente integrata nel contesto J2EE e permetta agli sviluppatori JSP di sfruttare — ponendosi in quest'ambito esattamente allo stesso livello di astrazione degli strati EJB e JDO — le connessioni preimpostate dall'Application Deployer in sede JNDI, condividendo in maniera ottimizzata con gli altri componenti della piattaforma Java Enterprise connessioni JDBC di qualsiasi natura e traendo vantaggio dalle funzionalità automatiche di connection pooling e distributed transactions della piattaforma J2EE.

Le funzionalità SQL vengono implementate tramite i seguenti tag:

- il tag <sql:query> permette di eseguire una query SQL (tipicamente tramite lo statement select) eventualmente parametrica restituendone il risultato in una variabile JSTL di tipo javax.servlet.jsp.jstl.sql.Result; il testo della query può essere fornito come attributo o come corpo del tag;

- il tag <sql:update> permette di inviare al server SQL un comando di aggiornamento (quali insert, update e delete); esattamente come per il tag precedente l'aggiornamento può avvenire in modalità parametrica e il testo del comando può essere specificato come attributo ovvero come corpo del tag; il risultato — in stile JDBC il numero di righe inserite, modificate o cancellate a seguito del comando — viene archiviato in una variabile JSTL;

- una serie di tag <sql:query> e <sql:update> possono essere innestati all'interno di un tag <sql:transaction> al fine di raggrupparli in una transazione; in base all'esito della transazione, il tag <sql:transaction> provvederà automaticamente al commit o alla rollback;

- eventuali parametri a query e aggiornamenti vengono forniti tramite tag <sql:param> innestati all'interno del relativo tag <sql:query> o <sql:update>.

Il risultato di una query viene fornito sotto forma di interfaccia javax.servlet.jsp.jstl.sql.Result, le cui principali proprietà sono:

- rows: rappresenta le righe della tabella risultato della query, sotto forma di SortedMap in cui le chiavi sono costituite dai nomi delle colonne e i valori dai valori delle rispettive colonne;
- rowCount: il numero di righe della tabella risultato;
- columnNames: i nomi delle colonne della tabella risultato, sotto forma di vettore di stringhe disposte secondo l'ordine con cui le colonne sono state ritornate dal server SQL.

Implementazione delle funzionalità di visualizzazione del traffico

Anche in questo caso le modifiche al codice Java sono pressoché inesistenti, e si limitano alla definizione di un nuovo dispatcher.

La prima sezione della view querytraffic.jsp, analogamente alla view querylogs.jsp, fornisce all'utente una ricapitolazione dei criteri di ricerca e ordinamento attuali, eventualmente impostando dei valori di default; successivamente viene eseguita la query, tenendo conto dei parametri di ricerca e di ordinamento impostati dall'utente:

```
<c:choose><c:when test="${!empty node}">
  <sql:query var="trafficData">
    select node_id,trd_date,trd_mbytes from traffic_data
    where node_id=? order by <c:out value="${order}" />
    <sql:param value="${node.id}" />
  </sql:query>
</c:when></c:choose>
```

Se l'utente non ha richiesto di restringere la ricerca a un nodo particolare, vengono lette tutte le righe dalla tabella di traffico; altrimenti la selezione viene limitata alle righe facenti capo al nodo specificato attraverso una query parametrica, la cui sintassi è assai simile a quella consueta in ambito JDBC.

In entrambi i casi, la selezione tiene conto dell'ordinamento impostato; si noti che, esattamente come in JDBC, l'ordinamento non può essere impostato come parametro, ma deve tradursi nella produzione di una query SQL personalizzata. In contesto JSTL ciò si traduce nella generazione di un tag <sql:query> a contenuto variabile tramite il tag <c:out>.

Il risultato della query è ora disponibile nella variabile trafficData, sulla cui proprietà rows è possibile iterare, ottenendo una a una le righe della nostra tabella risultato.

Ancora una volta il codice JSTL, alla cui visione si invita il lettore, si dimostra di facile comprensione. Si noti come, contestualmente alla lettura delle righe, i totali giornalieri di traffico vengano sommati nella variabile totalTraffic, inizializzata a 0 prima del tag <c:forEach> e visualizzata nell'ultima riga della tabella.

A piè di pagina viene quindi prodotto il form che permette all'utente di avviare una nuova ricerca; il codice relativo è simile a quello della view querytraffic.jsp, con la sola differenza che

questa volta la list box dei nodi è limitata ai soli router, l'unica categoria di dispositivi in grado di generare dati di traffico.

Internazionalizzazione e localizzazione

Per internazionalizzazione si intende l'astrazione di un'applicazione dalle caratteristiche culturali e geografiche delle piattaforme client.

Al processo di internazionalizzazione si accoppia invariabilmente la localizzazione, vale a dire l'adattamento — tipicamente a runtime — di un'applicazione alla caratteristiche di una specifica regione geografica.

Ai fini dell'internazionalizzazione si possono identificare due macrocategorie di informazioni.

In primo luogo, le informazioni dinamiche, per le quali l'internazionalizzazione consiste nello specificare la forma in cui esse vengono presentate all'utente; esempi tipici sono costituiti da numeri e date, la cui formattazione varia di Paese in Paese, mentre l'informazione in quanto tale rimane immutata.

Si hanno poi le informazioni statiche: in questo caso è l'informazione in sé ad essere funzione del processo di internazionalizzazione; ricadono in questa categoria tutti i messaggi di un'applicazione, che devono variare in ogni regione geografica in base alla lingua dell'utente.

L'insieme delle informazioni statiche e delle forme delle informazioni dinamiche che caratterizzano ogni regione geografica vengono denominate con il termine inglese *locale*. Si noti che, per un unico Paese, ovvero per un'unica lingua, possono essere definiti più *locale*. Questo è senz'altro il caso della lingua inglese, per la quale oltre all'atteso locale *en* (lingua inglese – regione geografica non specificata) esiste una serie di locale specifici, quali en_US (inglese – Stati Uniti d'America) o en_AU (inglese – Australia) e molti altri; ma anche della nostra lingua italiana, geograficamente molto meno diffusa dell'inglese, esiste oltre al locale it (italiano – Italia) è definito anche il locale it_CH (italiano – Svizzera).

L'internazionalizzazione di un'applicazione consiste nell'identificare e isolare tutte le sezioni di un'applicazione dipendenti dalla regione geografica del client, sostituendo a informazioni statiche e forme di presentazione dinamiche riferimenti a gruppi di risorse esterni, definiti bundle.

Nel mondo Java un bundle è costituito da un file di testo in formato ASCII, il cui nome è formato da un prefisso, usualmente legato all'applicazione e denominato *basename*, e dal nome del locale cui tale file si riferisce. Esso contiene un elenco di informazioni statiche, vale a dire di messaggi, secondo la seguente sintassi:

```
<chiave>=<valore>
```

dove <chiave> è il nome simbolico del messaggio, a cui l'applicazione si può riferire indipendentemente dalla piattaforma geografica di esecuzione, mentre <valore> è il contenuto del messaggio nel locale cui il file fa riferimento. In ambito Java usualmente le forme di presentazione dinamiche sono preconfigurate e vengono attivate automaticamente dal runtime in base alle caratteristiche del client.

In contesti web, la piattaforma geografica del client, vale a dire il locale, viene comunicata all'Application Server, e quindi all'applicazione, direttamente dal browser dell'utente (In Mozilla, Edit —> Preferences —> Navigator —> Languages; in Internet Explorer, Strumenti —> Opzioni Internet —> Lingue).

Funzionalità di internazionalizzazione di JSTL

L'ultimo componente di JSTL che ci rimane da analizzare è la libreria di internazionalizzazione, detta anche FMT o I18N.

Come per gli altri componenti, anche essa va specificata nel deployment descriptor e dichiarata tramite un elemento <%@ taglib> da ogni pagina JSP che intenda farne uso.

Il descrittore di deploy accetta tre parametri di configurazione, da definirsi come <context-param>, per la libreria FMT:

- javax.servlet.jsp.jstl.fmt.localizationContext: il basename del bundle utilizzato per default dall'applicazione;

- javax.servlet.jsp.jstl.fmt.fallbackLocale: il locale di default da utilizzare se un client non fornisce nessuna informazione a riguardo o se il locale richiesto dal client non è disponibile;

- javax.servlet.jsp.jstl.fmt.locale: questo parametro, di utilizzo raro, permette di forzare un locale ignorando effettivamente le informazioni di localizzazione fornite dal client.

I principali tag votati alle funzionalità di internazionalizzazione sono i seguenti:

- <fmt:message>: carica dal bundle specificato nell'attributo omonimo il messaggio la cui chiave è contenuta nell'attributo key, visualizzandolo o archiviandolo in una variabile JSTL; qualora l'attributo bundle non venga fornito, viene utilizzato il bundle corrente; la libreria FMT gestisce anche messaggi parametrici, vale a dire messaggi in cui una parte del testo può variare a runtime;

- <fmt:formatNumber>: formatta un numero in base alle caratteristiche del locale corrente e alla natura delle informazioni rappresentate dal numero stesso (numero, valuta o percentuale); tale tag dispone inoltre di diversi attributi che permettono di specificare il formato e la maschera di output delle informazioni;

- <fmt:parseNumber>: svolge l'azione opposta rispetto al tag precedente, effettuando il parsing di una stringa contenente un numero in formato localizzato e visualizzando o memorizzando il valore corrispondente in una variabile JSTL;

- <fmt:formatDate> e <fmt:parseDate>: in maniera analoga ai due tag precedenti consentono la formattazione e il parsing localizzato di date.

Implementazione delle funzionalità di internazionalizzazione

In virtù del carattere esemplificativo della nostra applicazione limiteremo l'internazionalizzazione alla sola funzionalità di visualizzazione dei dati del traffico; la procedura per le altre parti di NetView sarebbe in concreto esattamente identica.

Per maggior chiarezza introdurremo una nuova azione, querytrafficintl, che produce gli stessi report di traffico dell'azione querytraffic, ma in versione localizzata.

Come nei casi precedenti le uniche modifiche al codice Java sono costituite dalla definizione di un nuovo dispatcher e dalla sua dichiarazione nella classe FrontControllerServlet. Procediamo quindi all'identificazione delle risorse da internazionalizzare nella view querytrafficintl.jsp.

Partiamo dalle risorse dinamiche, per cui l'internazionalizzazione si applica alle forme di presentazione; esse sono costituite dai dati variabili, vale a dire nel nostro caso dalle date e dai totali di traffico giornalieri recuperati dal database SQL e dal totale di traffico visualizzato a piè di report; la visualizzazione di tali dati viene quindi racchiusa in tag <fmt:formatDate> e <fmt:formatNumber>.

Le risorse statiche, di cui dobbiamo internazionalizzare i contenuti, sono costituite da tutti i messaggi prodotti dalla nostra applicazione, che vanno estrapolati dalla view, tradotti in tutti i locale che si desidera supportare e archiviati in bundle separati; al posto dei messaggi statici si inseriscono nella view dei tag <fmt:message>.

A livello di deployment descriptor configuriamo come locale di fallback en_US e come basename di default per i bundle di NetView la stringa "messages".

Decidiamo di supportare i locale it ed en_US, creando i due bundle messages_it.properties e messages_en_US.properties, che archiviamo in WEB-INF/classes.

Si noti che, avendo definito un locale di fallback, abbiamo la certezza che ogni client della nostra applicazione sarà identificato da un locale; in caso contrario dovremmo definire un bundle per il locale nullo.

Le novità della view querytrafficintl.jsp rispetto alla view querytraffic.jsp consistono semplicemente nell'astrazione delle informazioni tramite i tag FMT di JSTL, vale a dire nella sostituzione dei messaggi statici con dei tag <fmt:message> e l'utilizzo degli elementi <fmt:formatDate> e <fmt:formatNumber> per la visualizzazione di date e numeri.

Nel primo caso gli attributi type e dateStyle, impostati rispettivamente a date e full, richiedono la visualizzazione dei soli dati di giorno, mese e anno in forma "lunga", vale a dire con nomi completi di giorno e mese.

I dati del traffico vengono invece presentati, secondo quanto specificato dall'attributo pattern, tramite un numero di cifre intere da 1 a 4 con eventuale separatore di migliaia. Tale informazione viene caratterizzata come semplicemente numerica ponendo l'attributo type al valore number; le altre possibilità sarebbero state currency e percentage, per indicare che il valore è da intendersi rispettivamente come valuta o percentuale.

Si noti infine che il processo di internazionalizzazione adottato, consistente nell'adozione dei bundle e nella completa astrazione di tutte le informazioni statiche, non è l'unico possibile.

Questo processo ha lo svantaggio di nascondere totalmente i contenuti reali delle pagine web ai grafici, complicando notevolmente le fasi di progettazione, impaginazione e manutenzione del layout HTML dell'applicazione.

Un'alternativa è data dalla realizzazione di versioni differenti delle pagine JSP in funzione dei diversi locale supportati, utilizzando una FrontControllerServlet esteso per ritornare al client una view specifica in base alle proprie caratteristiche geografiche.

Quest'impostazione, che privilegia il lavoro dei grafici e semplifica il lavoro di traduzione per applicazioni web caratterizzate da moli di testo significative, ha tuttavia l'effetto collaterale di produrre un elevato numero di view duplicate, complicando le fasi di manutenzione e aggiornamento del codice.

Nella pratica è consigliabile un approccio ragionato che si collochi a metà strada tra questi due estremi e che faccia uso delle funzionalità di inclusione parametriche di JSP per mantenere un insieme unico di view e isolare la gran parte delle informazioni testuali in file HTML esterni, direttamente gestibili dai grafici web, prevalentemente statici e duplicati per i diversi locale supportati.

Java Server Faces

Java Server Faces, il cui nome risulterà forse già noto ad alcuni lettori, è un framework Java per la realizzazione di interfacce HTML componentizzate, state-aware, riusabili ed event-driven, destinato con tutta probabilità a rappresentare uno standard in tema di Web User Interface. Al suo sviluppo collaborano tutti i grandi nomi dell'informatica Java-related — in prima fila Sun, IBM, Oracle, Fujitsu e Hewlett-Packard — assieme ai principali produttori di tool di sviluppo e attori della scena open-source quali BEA, Borland, Macromedia e l'Apache Group.

L'obiettivo consiste specificamente nella realizzazione di un framework per la realizzazione di interfacce utente web-based componentizzate, state-aware, espandibili, event-driven, indipendenti dal markup language utilizzato dalla piattaforma di fruizione e dal contesto di esecuzione. Di seguito si vedranno più nel dettaglio tali requisiti.

Per *componentizzazione* si intende la possibilità di considerare gli elementi dell'interfaccia alla stregua di unità di assemblaggio sulla scorta delle quali realizzare elementi GUI più complessi o intere interfacce utente. In particolare, le specifiche Java Server Faces si pongono l'obiettivo di risultare tool-friendly, in maniera da essere facilmente integrabili in ambienti di sviluppo RAD, consentendo agli sviluppatori di realizzare in breve tempo e con skill minimi interfacce utente per applicazioni web e distribuite con la stessa semplicità offerta da ambienti quali .NET e Delphi, semplicemente "collegando" elementi di interfaccia a componenti di gestione e di business logic lato server tramite catene di eventi.

Java Server Faces intende risolvere in maniera definitiva anche la questione della gestione dello stato nelle interfacce web, liberando lo sviluppatore dal bisogno di salvare e ripristinare lo stato di tutti i componenti di interfaccia utente ad ogni visualizzazione di pagina. L'obiettivo è quello di rendere gli elementi GUI "intelligenti", dotandoli della possibilità di validare in prima persona i dati inseriti dall'utente e di archiviare e ricaricare in maniera automatica e on demand il proprio stato da componenti/bean di memorizzazione lato server, denominati in gergo JSF *model object*, facili da realizzare e integrabili con architetture evolute quali JDO e EJB.

L'analisi dell'attuale stato dei framework di interfaccia utente in Java ha spinto i membri dell'Expert Group JSF a considerare l'espandibilità come un fattore imprescindibile: Java Server Faces si prefigge l'obiettivo di semplificare le possibilità di personalizzazione di rendering e di comportamenti, così come la creazione di elementi di interfaccia utenti complessi e di utilizzo generale; l'obiettivo è fornire una specifica e un insieme di componenti GUI di primo livello, lasciando ai produttori di tool la possibilità di personalizzare ed espandere l'offerta di elementi di interfaccia utilizzabili in applicazioni web e distribuite.

Insieme alla questione dello stato, uno dei punti cardine di JSF è rappresentato dalla definizione di un nuovo paradigma di event handling, che avvicini la programmazione in ambito web agli idiomi di gestione asincrona e discreta degli eventi già noti agli sviluppatori in ambienti GUI

client-server, sfruttando quando disponibili le capacità della piattaforma di fruizione per localizzare presso il client il processing di determinati eventi.

L'indipendenza dal markup language (HTML, DHTML, XML) viene ottenuta grazie alla completa separazione tra forma e comportamento dei componenti GUI: ogni modello di interazione lato server (quali "seleziona un elemento da una lista", "permetti all'utente di inserire manualmente dei dati") viene realizzato a livello client tramite dei renderer diversificati, i quali producono un'interfaccia utente in grado di soddisfare i requisiti funzionali del server al meglio delle possibilità della piattaforma di fruizione dell'utente.

La nozione di indipendenza dal markup language, ossia dal contesto lato client, viene estesa anche all'ambito lato server, il quale — al fine di garantire la massima longevità di Java Server Faces — è indipendente dallo stesso paradigma Java Server Pages ed è stato progettato direttamente e unicamente in funzione delle Servlet API.

La più recente Early Access Release di Java Server Faces consente di farci un'idea sufficientemente precisa dell'architettura di questa API, che "voci di corridoio" danno come destinata a essere finalizzata nella sua versione 1.0 a breve, per essere utilizzata in progetti di test.

Si noti infine che Java Server Faces sfrutta internamente diverse API Java open source rilasciate in seno al gruppo Apache e divenute ormai standard *de facto*. Fra queste va sicuramente menzionato Struts, forse il framework Java generalista di maggior successo in assoluto, verso il quale JSF si propone espliciti obiettivi di integrazione e nei cui confronti — con particolare riferimento al package `struts-html`, i cui tag sono destinati ad andare in overlapping con Java Server Faces — intende fornire espliciti tool di migrazione.

L'architettura di Java Server Faces

Chiarito in generale il contesto in cui Java Server Faces si colloca e gli obiettivi che ne caratterizzano il design, passiamo all'analisi strutturale di questo framework.

L'attuale contesto di utilizzo di JSF è all'interno di applicazioni J2EE, in qualità di layer deputato all'implementazione di interfacce utente web/Internet.

Nella sua attuale implementazione Java Server Faces è in effetti una tag library di JSTL, grazie al quale risulta perfettamente integrato con JSP e con tutti i protocolli e API del mondo Java Enterprise, che le nostre applicazioni JSF potranno sfruttare trasparentemente.

Un'interfaccia utente realizzata in JSF consisterà dunque in un insieme di pagine JSP, al cui interno utilizzeremo i tag Java Server Faces per disegnare elementi di interfaccia utente, e in una pluralità di oggetti Java delegati a rappresentare e validare lo stato dei suddetti componenti e a gestire il processing degli eventi, le funzionalità di navigazione e il ciclo di vita dell'applicazione.

I tag Java Server Faces verranno tradotti in elementi di interfaccia utente solamente a runtime e in funzione delle caratteristiche e possibilità della piattaforma di fruizione dell'utente.

I quattro elementi fondanti di JSF, che analizzeremo più nel dettaglio nel corso della nostra esplorazione, sono:

- i componenti di interfaccia utente lato server;
- gli elementi di interfaccia utente lato client;

- i model object, vale a dire i bean — accoppiati ai componenti di interfaccia utente — che ne rappresentano e memorizzano lo stato;

- l'oggetto ApplicationHandler, ossia il gestore di eventi globali dell'applicazione JSF.

I componenti di interfaccia utente lato server

Molti, leggendo il titolo di questo paragrafo, avranno già notato l'ossimoro: come può un componente di interfaccia utente, vale a dire qualcosa che per definizione è strettamente legato al client, esistere "lato server"?

La risposta è che JSF mantiene lato server la conoscenza dei "modelli di interazione" di un'interfaccia utente, vale a dire quell'insieme di comportamenti, azioni e reazioni — gestione degli eventi, dialogo con i model object per la gestione dello stato e con gli elementi UI lato client per il rendering dei componenti — che identificano in maniera univoca un elemento di interfaccia utente indipendentemente dalle sue peculiarità di rappresentazione visiva.

I componenti UI lato server sinora presenti in JSF sono sette e rappresentano gli elementi base comuni alla grande maggioranza delle interfacce utente. Essi sono destinati a essere ampliati, sia direttamente a livello di specifiche Java Server Faces sia tramite librerie di componenti di terze parti:

- UIOutput: rappresenta un insieme di informazioni testuali da produrre in output;

- UITextEntry: questo modello di interazione rappresenta un elemento di interfaccia tramite il quale l'utente può inserire delle informazioni in formato testuale;

- UISelectBoolean: consente all'utente di fornire un'informazione di tipo booleano;

- UISelectOne: permette all'utente di selezionare un elemento all'interno di una lista a valori predefiniti;

- UICommand: rappresenta un elemento di interfaccia in grado di generare un evento all'atto della sua "attivazione" da parte dell'utente;

- UIForm: questo componente UI lato server raggruppa un insieme di controlli logicamente correlati in una sorta di "unità transazionale" e è affine al tag <form> in HTML;

- UIPanel: raggruppa un insieme di elementi di interfaccia utente al fine di coordinarne la modalità di presentazione a video, svolgendo un ruolo paragonabile ai Layout Manager in ambiente Swing.

Gli elementi di interfaccia utente lato client

Ogni modello di interazione lato server, per poter essere effettivamente utilizzato, deve essere associato a uno o più elementi di interfaccia utente, selezionati in base alle caratteristiche della piattaforma client e compatibili con lo schema di comportamento desiderato.

Tabella 6.2 – *Componenti del RenderKit HTML.*

Componente lato client	Funzionalità	Rendering
output_date / time / number / text	Visualizza data / ora / numero / testo	Testo formattato
textentry_input / textarea / secret	Input di stringhe e password	Campo di input single-line / multi-line / no-echo
selectboolean_checkbox	Input di un valore booleano	Checkbox
selectone_radiogroup / optionlist	Selezione di un elemento da una lista	Gruppo di bottoni radio / list-box
command_button / hyperlink	Submit di form / URL link	Bottone / hyperlink
form	Raggruppa in un'unità logica un insieme di componenti	Nessuna rappresentazione
panel_list / group	Visualizza i dati contenuti in un array, Iterator, Collection o Map	Dati in forma tabellare

Tali gruppi di elementi GUI lato client vengono in gergo JSF denominati RenderKit e costituiscono, in ottica Model–View–Controller, i componenti view di JSF.

L'implementazione Early Access Release 4 di JSF fornisce un RenderKit in HTML standard, ed è ragionevole presumere che nuovi RenderKit verso ulteriori linguaggi di markup (XML, DHTML, light-HTML o WAP per dispositivi wireless) verranno realizzati da produttori indipendenti successivamente al rilascio definitivo di Java Server Faces.

Si noti che per ogni modello di interazione lato server possono esistere una o più rappresentazioni a livello di interfaccia utente, denominate Renderer. In ogni tag JSF vengono quindi specificati sia il modello di interazione lato server richiesto sia il Renderer desiderato. Questa separazione di ruoli permette di adattare facilmente la presentazione della web UI alle caratteristiche della piattaforma client di fruizione e di mantenere effettivamente separati i compiti degli autori di componenti da quelli dei grafici web e degli sviluppatori di applicazioni.

Il suddetto HTML RenderKit fornisce un ampio numero di componenti, in grado di realizzare tutti i modelli di interazione lato server precedentemente esaminati. Analizziamo i principali, riportati in tab. 6.2.

Si noti come il nome di ogni componente lato client identifichi in maniera univoca il modello di interazione lato server cui esso è associato.

La lista appena fornita, che rappresenta comunque un sottoinsieme dei componenti disponibili, è destinata a espandersi in maniera significativa in occasione del rilascio definitivo di Java Server Faces con l'implementazione di funzionalità più generiche di input e di output, di rappresentazione di immagini e di selezioni multiple da un ventaglio di possibilità.

Una delle caratteristiche di maggiore interesse del framework JSF è inoltre dato dalla sua estensibilità, la quale permette agli sviluppatori di estenderne le funzionalità attraverso la modellazione di nuovi schemi di interazione lato server e la codifica dei corrispondenti componenti lato client.

I model object

Un elemento fondamentale dell'architettura Java Server Faces è dato da quegli oggetti cui spetta la responsabilità di modellare e gestire le informazioni di stato associate agli elementi di interfaccia. Tali oggetti, denominati model object in gergo JSF, vengono implementati come oggetti JavaBean property-based.

La flessibilità nell'accoppiamento tra le property dei model object realizzati dallo sviluppatore e i componenti di interfaccia viene garantita dai meccanismi di JSF, i quali permettono il libero utilizzo di stringhe, tipi numerici, tipi data/ora, vettori, Collection, Iterator e Map nel ruolo di classi di back-end.

Rivestono poi un ruolo importante nel contesto dei model object e del loro dialogo con i diversi componenti di un'applicazione i meccanismi di validazione e di data conversion forniti da Java Server Faces.

Tale framework dispone infatti di strutture automatizzate rivolte alla validazione dello stato dei componenti prima ancora che esso venga notificato ai model object e basate sulla famiglia di tag <validate_*>, i quali permettono di vincolare valori di diversi tipi in base ai loro contenuti e al loro stato.

Il criterio di estensibilità che domina le specifiche di JSF si applica anche ai componenti di validazione e permette agli sviluppatori di definire nuovi criteri di verifica e controllo preventivo dei dati tramite l'implementazione di classi e tag appositi.

Discorso analogo vale per le funzionalità di conversione, intese come quei meccanismi che regolano il flusso dei dati tra model object e componenti di presentazione. Tipicamente i primi manterranno le informazioni di stato in oggetti semanticamente affini ai dati da modellare, mentre i secondi avranno l'esigenza di convertire — tanto in lettura quanto in scrittura — tali informazioni in formati di presentazione facilmente comprensibili e gestibili dall'utente.

Java Server Faces fornisce agli sviluppatori numerosi meccanismi automatici di conversione, tramite i quali risulta possibile delegare il processo di conversione tra view e model layer direttamente a JSF. In particolare sono supportate in maniera nativa le conversioni da e verso tipi numerici, java.util.Date, java.util.Time e java.util.DateTime a partire da stringhe formattate in logica java.text.DateFormat, java.text.TimeFormat e java.text.NumberFormat.

Per far fronte a necessità particolari JSF supporta infine la definizione e implementazione di meccanismi di conversione ad hoc attraverso la codifica di appositi oggetti Converter.

L'oggetto ApplicationHandler e la gestione degli eventi

Nell'ottica del rispetto degli standard esistenti e della minimizzazione dei tempi di apprendimento, la gestione degli eventi in contesto JSF segue schemi già collaudati e familiari agli sviluppatori Java e è basata sull'usuale combinazione di classi Listener e Event.

In risposta alle azioni dell'utente i componenti generano degli eventi, dei quali un'applicazione Java Server Faces può essere notificata tramite la registrazione di adeguati oggetti Listener.

Il framework Java Server Faces conosce tre classi di eventi:

Eventi di tipo value-changed

Vengono generati all'atto della modifica dello stato di un componente di tipo UITextEntry, UISelectOne o UISelectBoolean e gestiti da oggetti di classe ValueChangedListener; un classico esempio è dato dalla selezione di una checkbox da parte di un utente.

Eventi di tipo action

I componenti UICommand generano un evento di questa classe in seguito alla loro attivazione da parte dell'utente, tipicamente tramite un click su di un bottone o di un hyperlink; gli eventi di questa classe vengono gestiti da oggetti ActionListener.

Eventi a livello di applicazione

Sostanzialmente analoghi agli action event, vengono generati in seguito ad un'azione i cui effetti si applicano ad un intero albero di componenti. Un tipico esempio è dato dal click sul bottone submit di un form, che provoca l'aggiornamento dello stato di tutti i componenti innestati.

Particolarmente importanti sono gli eventi dell'ultimo tipo, i cui effetti si ripercuotono sull'intero stato dell'applicazione. Essendo per loro stessa natura globali, la gestione di tali eventi è delegata ad un'apposita classe Singleton che implementa l'interfaccia javax.faces.lifecycle .ApplicationHandler.

Tale classe, componente necessario di ogni sistema software basato su Java Server Faces, ha il ruolo di recepire gli application event, di coordinarne l'impatto a livello globale — sincronizzandoli se necessario con lo stato dei componenti — e di gestire la navigazione dell'utente all'interno dell'applicazione.

Java Server Faces e NetView

Esaminati i componenti fondamentali e l'architettura generale di JSF siamo in grado di applicare le nostre nuove conoscenze a un esempio pratico. Supporremo infatti che il committente di NetView ci abbia richiesto di dotare la nostra web application di funzionalità di configurazione relative alla specifica di alcuni parametri globali che influenzano il comportamento dell'applicazione, funzionalità che intendiamo realizzare tramite un form basato su Java Server Faces. La nostra implementazione si articolerà su quattro punti principali:

- l'integrazione delle funzionalità di configurazione all'interno di NetView;
- la realizzazione dello strato view — ossia la web UI — relativo al form di configurazione;
- l'implementazione dei model object delegati a rappresentare e modellare le informazioni di configurazione;
- la codifica dell'oggetto ApplicationHandler — responsabile del trattamento degli eventi a livello di applicazione e della user navigation — e la gestione del suo ciclo di vita.

La nostra web UI Java Server Faces dovrà in particolare permettere la gestione di varie informazioni. Innanzitutto, due descrizioni del sistema software NetView — la prima concisa, la seconda più dettagliata — per le quali utilizzeremo rispettivamente i componenti textentry_input e textentry_textarea, vale a dire un campo di input single-line e un'area di input multilinea.

Poi, una master password SNMP, che rappresenteremo tramite un componente textentry_secret, che implementa in maniera nativa la semantica richiesta (campo di input single-line e no-echo).

Quindi, l'impostazione della modalità di log, che realizzeremo attraverso il componente selectone_radiogroup, il quale implementa la semantica della mutua esclusione dei valori possibili tramite un gruppo di bottoni radio.

È necessaria anche la configurazione di un limite giornaliero di traffico, gestito tramite un normale campo di input textentry_input; in questo caso sfrutteremo le funzionalità di validazione di Java Server Faces per impedire all'utente di proporre soglie di traffico inferiori al limite dei 100 MB giornalieri.

Poi, la selezione della notifica all'amministratore di sistema, che modelleremo grazie al componente selectone_optionlist, tradotto dal Renderer in una drop-down listbox.

Infine, rappresenteremo le informazioni testuali da fornire all'utente all'interno della pagina JSF come componenti output_text.

Provvederemo a coagulare le suddette informazioni in una pagina JSF composta da un form, ad ogni elemento del quale corrisponderanno un parametro di configurazione e una property del model object models.ConfigurationBean.

In un contesto reale alla gestione del form di configurazione si accompagnerebbe la codifica dei relativi comportamenti all'interno di NetView; nel nostro esempio, focalizzato unicamente sullo strato di interfaccia utente, tralasceremo tali attività implementative. Parimenti non ci porremo il problema della persistenza delle informazioni di configurazione, che verranno semplicemente mantenute nel model object deputato, andando perdute allo shutdown dell'applicazione.

Integrazione in NetView

L'integrazione delle nuove funzionalità all'interno della nostra applicazione consiste semplicemente nella definizione di una nuova azione config a livello di FrontControllerServlet e nella definizione di un Dispatcher, ConfigDispatcher, la cui unica attività consiste nell'invocare la view config.jsp contenente la definizione della web UI del form di configurazione in formato JSF.

Il secondo e ultimo passo dell'integrazione in NetView delle nuove funzionalità consiste nella modifica del deployment descriptor secondo quanto richiesto dalle attuali specifiche Java Server Faces, quindi definizione della classe listener NetViewServletContextListener, la cui funzione consiste nel gestire il ciclo di vita dell'oggetto ApplicationHandler, e attivazione delle funzionalità Java Server Faces attraverso la dichiarazione del Faces Servlet e sua assegnazione algli URL /faces/*.

La web UI

L'interfaccia utente che implementa le funzionalità di configurazione richieste dal cliente è contenuta nella view config.jsp — cui si invita il lettore a fare continuo riferimento per tutto il prosieguo di questa sezione — la quale dichiara la propria intenzione di far uso di JSF tramite l'utilizzo del tag <%@ taglib>.

All'interno di tale pagina JSP il model object models.ConfigurationBean associato ai componenti UI Java Server Faces utilizzati da NetView viene istanziato come un normale JavaBean tramite una direttiva <useBean>. Segue quindi l'apertura del tag <faces:usefaces>, chiuso al termine della view appena prima dell'inclusione di bottombar.jsp; all'interno di tale tag è possibile utilizzare liberamente i componenti JSF.

I nostri elementi di interfaccia JSF vengono raggruppati in un unico form — denominato configForm — al cui interno provvediamo a emettere le necessarie informazioni testuali destinate all'utente con dei tag di tipo <faces:output>.

Tali informazioni si sarebbero ovviamente potute presentare a video in maniera più diretta; tuttavia, codificando ogni messaggio in un apposito tag Java Server Faces eseguiamo quell'attività di formalizzazione dell'output che ci permetterà in futuro di implementare con facilità sofisticate funzionalità di internazionalizzazione e localizzazione.

Segue quindi la rappresentazione dei parametri di configurazione, che viene effettuata con la massima semplicità attraverso gli elementi di interfaccia utente selezionati in fase di analisi. Si noti che, grazie all'esplicita associazione con le relative property dei model object, lo sviluppatore è completamente sgravato dal compito di gestire lo stato dell'applicazione web. JSF sincronizzerà automaticamente componenti view e model object in risposta alle azioni dell'utente.

L'implementazione dei model object

I model object rappresentano lo stato dell'applicazione web e hanno il compito di fornire servizi di persistenza ai componenti di web UI JSF. Nel nostro caso sarà sufficiente codificare un unico model object, models.ConfigurationBean, delegato a rappresentare i parametri di configurazione impostati dall'utente attraverso la view config.jsp; come è lecito attendersi si tratterà di un JavaBean le cui proprietà corrisponderanno perfettamente agli elementi di interfaccia utente analizzati nel paragrafo precedente, la cui implementazione — sorgente alla mano — vogliamo ora commentare.

La codifica delle descrizioni di NetView, della password SNMP e della soglia di traffico è elementare e consta semplicemente di quattro membri privati supportati dai rispettivi accessori.

È da rilevare come nessuna operazione esplicita ci venga richiesta per la gestione del tipo della property maxDailyTraffic: JSF si prende automaticamente cura della conversione in intero di quanto inserito dall'utente nel form di input.

Altrettanto semplice è la gestione della property alarmMode, in cui verrà memorizzata la modalità di notifica all'amministratore; essa viene infatti codificata come una semplice string property, in maniera assolutamente identica ad appTitle, appDesc e snmpPassword. Il nostro model object è infatti completamente ignaro dell'esistenza di una lista di valori ammissibili per questa proprietà, funzionalità che viene integralmente gestita da Java Server Faces dietro le quinte.

Appena più complicata è l'implementazione della property associata alla modalità dei log. In questo caso, infatti, abbiamo scelto di rendere le possibili impostazioni di questa proprietà parte integrante del model object.

Provvediamo quindi a modellare tramite un'elementare non-mutable data class la nozione di "modalità di log" come insieme di due valori stringa, pari rispettivamente al codice della classe di log e alla sua rappresentazione visuale (descrizione), dotando quindi il nostro model object della conoscenza di tutte le modalità di log supportate.

Nel costruttore del model object provvediamo alla conversione del vettore di oggetti LogMode nel formato gestito da Java Server Faces, vale a dire una lista di oggetti javax.faces.component.SelectItem.

Segue quindi l'implementazione della property logMode come semplice string property; Java Server Faces — sulla scorta degli oggetti SelectItem da noi specificati — provvederà in maniera autonoma a gestire l'interazione con l'utente fornendo direttamente al model object models.ConfigurationBean il codice della modalità di log impostata via web UI.

Infine ci assicuriamo di fornire un default valido per la modalità di log, specificando un opportuno valore all'atto della creazione del model object.

L'oggetto ApplicationHandler

Come già fatto notare, ogni Web Application in ambiente Java Server Faces deve possedere una classe singleton che implementi l'interfaccia javax.faces.lifecycle.ApplicationHandler, la cui responsabilità consiste nel dirigere la navigazione dell'utente all'interno dell'applicazione in risposta agli eventi di tipo application level. Nel nostro caso l'implementazione di tale classe sarà particolarmente semplice, non dovendo gestire la nostra applicazione che un unico evento globale, corrispondente alla conferma da parte dell'utente dei dati inseriti nel form di configurazione.

Il fulcro di un oggetto ApplicationHandler è rappresentato dal suo metodo processEvent() che — in logica vagamente JDK 1.0 — viene notificato allo scattare di qualsiasi evento. Esso provvede quindi ad identificare l'evento ricevuto e, in base allo stato dell'applicazione, aggiorna i componenti del sistema e guida se necessario la navigazione dell'utente all'interno delle view JSF/JSP.

È responsabilità dello sviluppatore gestire il ciclo di vita dell'oggetto ApplicationHandler, istanziandolo alla creazione del ServletContext dell'applicazione e rilasciandolo alla sua distruzione. Questo ruolo viene svolto in NetView dalla classe NetViewServletContextListener, che abbiamo visto venire dichiarata come listener a livello di deployment descriptor.

Custom Tag Library

Per Custom Tag Library — in breve CTL — si intende un gruppo di classi Java che, unitamente a un apposito document descriptor, implementano un'estensione della sintassi JSP consistente in uno o più tag XML-like.

Tali tag donano allo sviluppatore accesso a un insieme di nuove funzionalità e di comportamenti originali che egli può utilizzare e combinare in maniera libera e creativa con altre Tag Library all'interno delle proprie pagine JSP.

Il meccanismo delle CTL — introdotto a partire dalla versione 1.1 di Java Server Pages — è estremamente flessibile e generico, e garantisce una potenza espressiva sostanzialmente illimitata, come testimoniato dall'assoluta varietà strutturale e eterogeneità funzionale delle Custom Tag Library disponibili sul mercato.

Struttura di una Custom Tag Library

Una Custom Tag Library è costituita da tre componenti principali:

- un insieme di classi Java dedicate all'implementazione delle funzionalità dei tag, denominate *tag handler*;
- un *tag library descriptor*, vale a dire un documento XML contenente le specifiche funzionali della CTL;
- un sistema opzionale di validazione — rappresentato da una o più classi Java — il cui compito consiste nel verificare la correttezza sintattica delle pagine JSP che utilizzeranno la CTL.

Ai punti sopra menzionati si affiancano gli importanti temi del deployment, della gestione degli attributi, del tag nesting, delle eccezioni e degli eventi.

I tag handler

Per tag handler si intende una classe Java cui spetta il compito di fornire l'implementazione dei comportamenti associati a uno specifico tag al JSP engine corrente e di comunicare con esso nel rispetto di una sequenza precisa di callback. Si noti che alle classi tag handler — diversamente da quanto accade per le servlet — non viene richiesto di essere rientranti: infatti, anche se il motore JSP ha il diritto di riutilizzare oggetti tag handler già istanziati, in ogni caso esso non può servirsi del medesimo tag handler per processare due tag contemporaneamente.

Java Server Pages 1.2 identifica tre categorie di tag handler — cui corrispondono altrettante interfacce definite nel package `javax.servlet.jsp.tagext` — differenziate in base alla complessità e alla natura delle operazioni che svolgono sul contenuto delle pagine JSP: i tag handler base, i quali implementano l'interfaccia `javax.servlet.jsp.tagext.Tag`, i tag handler iterativi — associati all'interfaccia `javax.servlet.jsp.tagext.IterationTag` — e i tag handler di contenuto, vincolati all'interfaccia `javax.servlet.jsp.tagext.BodyTag`.

I tag handler base

I tag handler base sono caratterizzati da un'interazione ridotta con la pagina JSP al cui interno agiscono; in particolare essi non effettuano alcuna manipolazione del loro body content, vale a dire del codice HTML/JSP contenuto all'interno dei rispettivi tag di apertura e chiusura. Tali tag handler, come già visto, implementano l'interfaccia `javax.servlet.jsp.tagext` e sono caratterizzati da un meccanismo di callback estremamente semplice e basato sui metodi `doStartTag()` e `doEndTag()`, che andiamo ad analizzare.

```
int doStartTag() throws javax.servlet.jsp.JspException
```

`doStartTag()` viene invocato ad ogni utilizzo all'interno di una pagina JSP del tag rappresentato dal tag handler e comunica con il motore JSP tramite il suo valore di ritorno:

- `javax.servlet.jsp.tagext.Tag.EVAL_BODY_INCLUDE`: il JSP engine procede alla normale valutazione del corpo del tag e di ogni eventuale subtag;

- `javax.servlet.jsp.tagext.Tag.SKIP_BODY`: il corpo del tag viene interamente ignorato dal motore JSP, che salta direttamente al relativo tag di chiusura.

È anche possibile per il metodo `doStartTag()` generare un'eccezione figlia di `javax.servlet.jsp.JspException`, al fine di segnalare una condizione di errore grave: in questo caso il JSP engine interromperà il processo di valutazione dell'intera view JSP, usualmente ritornando al browser una pagina d'errore.

```
int doEndTag() throws javax.servlet.jsp.JspException
```

`doEndTag()` viene invocato in corrispondenza del tag di chiusura sull'oggetto tag handler corrispondente al tag corrente. Anche in questo caso la comunicazione con il motore JSP è affidata ad un semplice valore di ritorno:

- javax.servlet.jsp.tagext.Tag.EVAL_PAGE: il JSP engine procede normalmente al processing del resto della pagina;

- javax.servlet.jsp.tagext.Tag.SKIP_PAGE: la valutazione della pagina corrente viene interrotta.

Il ciclo di vita di un tag handler consiste dunque in coppie di invocazioni dei suddetti metodi doStartTag() e doEndTag(), seguiti da una chiamata finale al metodo release(), in risposta al quale il tag handler è tenuto a rilasciare immediatamente tutte le risorse in proprio possesso e a predisporsi alla finalizzazione. Si noti che, all'atto dell'invocazione di doStartTag(), il tag handler può fare affidamento sulla corretta inizializzazione del proprio contesto, consistente nell'esecuzione preventiva da parte del JSP engine dei seguenti passi:

- la proprietà pageContext del tag handler viene impostata, tramite l'apposito metodo setPageContext() dell'interfaccia javax.servlet.jsp.tagext.Tag, all'oggetto contenente le informazioni di contesto relative alla pagina JSP corrente;

- in caso di tag innestati la proprietà parent del tag handler punta al relativo tag handler padre;

- qualora al tag siano associati degli attributi essi vengono impostati correttamente nell'istanza di tag handler.

Figura 6.1 – *Ciclo di vita di un tag handler base.*

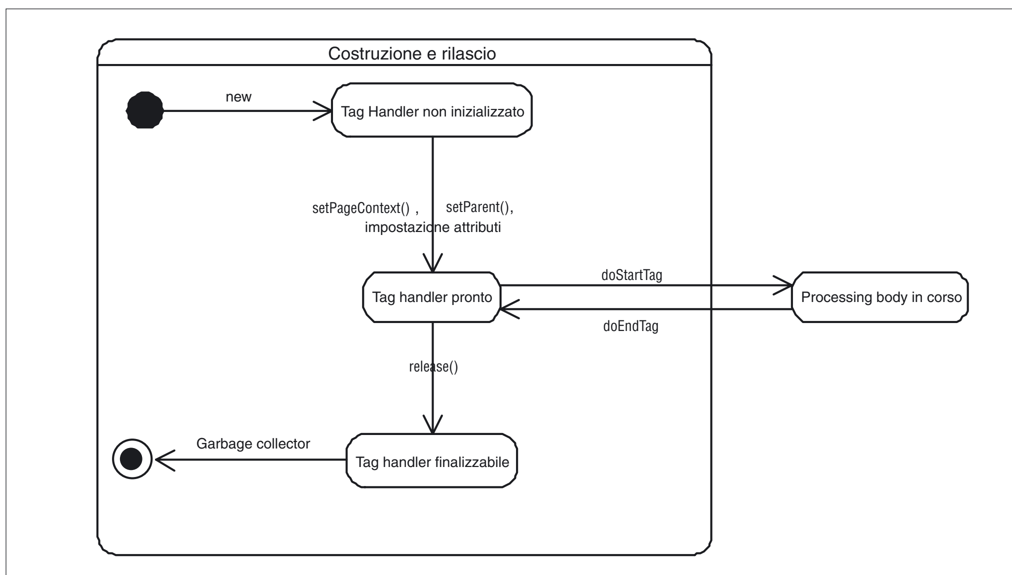
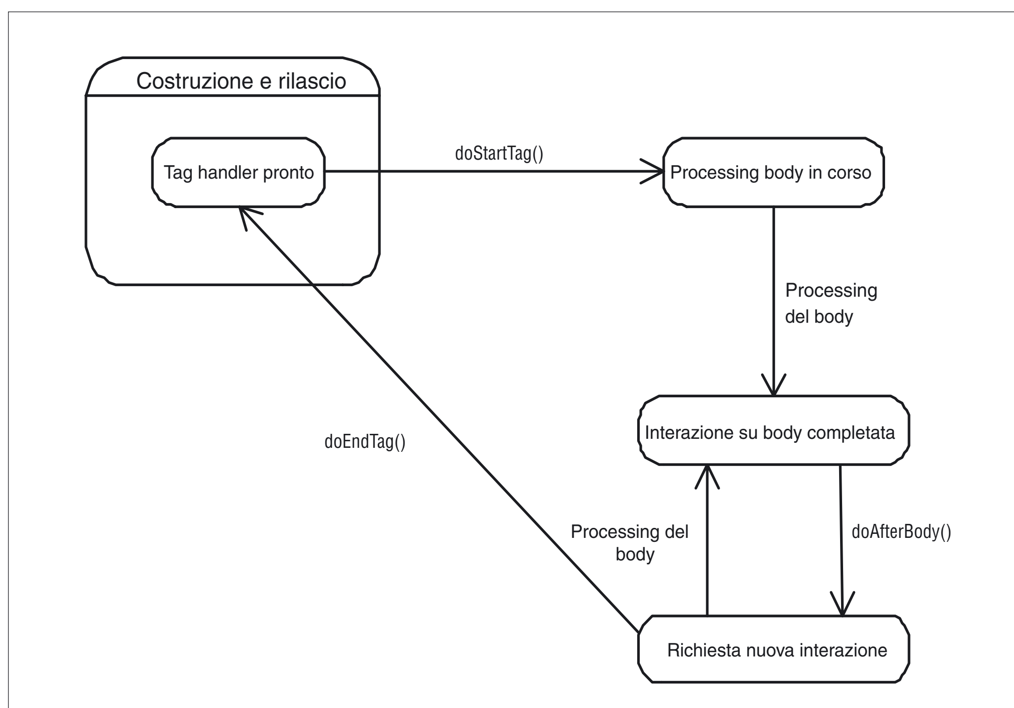


Figura 6.2 – *Ciclo di vita di un tag handler iterativo.*



Dal momento che buona parte dei metodi definiti nell'interfaccia Tag viene di norma codificata in maniera identica in ogni tag handler, il package javax.servlet.jsp.tagext mette a disposizione la convenience class TagSupport, la quale fornisce un'implementazione standard di ogni metodo d'interfaccia. Definire i propri tag handler base a partire dalla classe TagSupport permette agli sviluppatori di evitare la codifica diretta degli accessori per le proprietà parent e pageContext, nonché tipicamente l'implementazione del metodo release(), consentendo loro di concentrarsi sui soli metodi doStartTag() e doEndTag().

I tag handler iterativi

L'interfaccia javax.servlet.jsp.tagext.IterationTag rappresenta una delle novità della versione 1.2 del framework Java Server Pages e consente un'ìmplementazione estremamente semplificata di tag iterativi, vale a dire di quei tag che hanno bisogno di valutare ripetutamente il proprio body content sul modello del tag <c:forEach> di JSTL.

Tale interfaccia estende javax.servlet.jsp.tagext.Tag, aggiungendovi un unico metodo:

```
int doAfterBody() throws javax.servlet.jsp.JspException
```

doAfterBody() viene invocato al termine di ogni valutazione del contenuto del tag — dopo il doStartTag() iniziale e prima del doEndTag() finale — e comunica con il motore JSP tramite il suo valore di ritorno:

- javax.servlet.jsp.tagext.IterationTag.EVAL_BODY_AGAIN: il motore JSP effettuerà un'altra iterazione sul tag, valutandone nuovamente il body e gli eventuali subtag;

- javax.servlet.jsp.tagext.Tag.SKIP_BODY: la valutazione di questo tag è da ritenersi conclusa e il JSP engine può procedere al processing del resto della pagina.

Anche nel caso dei tag handler iterativi è possibile per lo sviluppatore risparmiare parte del lavoro di codifica definendo la propria classe come child class di javax.servlet.jsp.tagext.TagSupport; sarà quindi sufficiente, nella gran parte dei casi, limitarsi all'implementazione dei metodi doStartTag(), doAfterBody() e doEndTag().

I tag handler di contenuto

Un'interazione più complessa caratterizza i tag handler di contenuto, rappresentati da classi che implementano l'interfaccia javax.servlet.jsp.tagext.BodyTag. Tale interfaccia — basata su javax.servlet.jsp.tagext.Tag nella versione 1.1 di Java Server Pages — in JSP 1.2 estende invece javax.servlet.jsp.tagext.IterationTag e fornisce un insieme di metodi addizionali rivolti alla gestione del body content di un tag.

Il metodo doStartTag() supporta nel caso di tag handler di contenuto un insieme esteso di valori di ritorno:

- javax.servlet.jsp.tagext.Tag.EVAL_BODY_INCLUDE: il JSP engine procede alla normale valutazione del corpo del tag e di ogni eventuale subtag;

- javax.servlet.jsp.tagext.Tag.SKIP_BODY: il corpo del tag viene interamente ignorato dal motore JSP, che salta direttamente al relativo tag di chiusura;

- javax.servlet.jsp.tagext.BodyTag.EVAL_BODY_BUFFERED: laddove le funzionalità dei due precedenti codici di ritorno corrispondono perfettamente a quanto visto nel caso dei tag handler base, il valore EVAL_BODY_BUFFERED permette la gestione in modalità bufferizzata del risultato della valutazione del body content del tag.

Si noti che a partire dalla versione 1.2 del framework Java Server Pages l'utilizzo del valore di ritorno javax.servlet.jsp.tagext.BodyTag.EVAL_BODY_TAG, impiegato in JSP 1.1 — in funzione del contesto — con la medesima semantica di EVAL_BODY_AGAIN o di EVAL_BODY_BUFFERED è ufficialmente sconsigliato.

In caso di ritorno del valore EVAL_BODY_BUFFERED il motore JSP provvede a invocare immediatamente sul tag handler due metodi aggiuntivi definiti nell'interfaccia javax.servlet.jsp.tagext.BodyTag, indicati di seguito.

```
void setBodyContent(BodyContent bodyContent)
```

A setBodyContent() spetta semplicemente la responsabilità di impostare la proprietà bodyContent del tag handler al buffer utilizzato per memorizzare il contenuto del tag. Tale buffer viene rappresentato da un'istanza della classe javax.servlet.jsp.tagext.BodyContent, child class di javax.servlet.jsp.JspWriter.

```
int doInitBody() throws javax.servlet.jsp.JspException
```

Questo metodo viene invocato per dare al tag handler la possibilità di effettuare specifiche attività di inizializzazione in seguito all'impostazione del body content e di interrompere, in caso di condizione di errore critico, il processing dell'intera pagina corrente.

Nel ciclo di vita di un tag handler di contenuti seguono quindi una o più chiamate al metodo doAfterBody() — la cui semantica è identica a quella già analizzata per l'interfaccia IterationTag — e una chiamata finale al metodo doEndTag(), con la quale si conclude il processing del tag iniziato con l'invocazione del metodo doStartTag(). L'ultimo atto nella vita di un tag handler di contenuti è rappresentato dalla chiamata da parte del JSP engine del suo metodo release(), a fronte del quale il tag handler è tenuto a rilasciare tutte le risorse a propria disposizione, predisponendosi per la garbage collection.

Qualora il tag handler abbia richiesto in sede di doStartTag() un processing buffer-based dei contenuti del tag tramite il valore di ritorno EVAL_BODY_BUFFERED è sua diretta responsabilità provvedere all'output del body content tramite il Writer associato, operazione tipicamente svolta nel metodo doEndTag().

Grazie a questa gestione dell'output in modalità lazy risulta possibile per il tag handler — in qualsiasi momento del suo ciclo di vita — applicare nuove trasformazioni al body content o addirittura sopprimerlo interamente semplicemente evitando di inviarlo al Writer.

Anche nel caso dei tag handler di contenuti è possibile per gli sviluppatori utilizzare implementazioni predefinite per diversi metodi d'interfaccia definendo la propria classe a partire dalla convenience class javax.servlet.jsp.tagext.BodyTagSupport.

Il Tag Library Descriptor

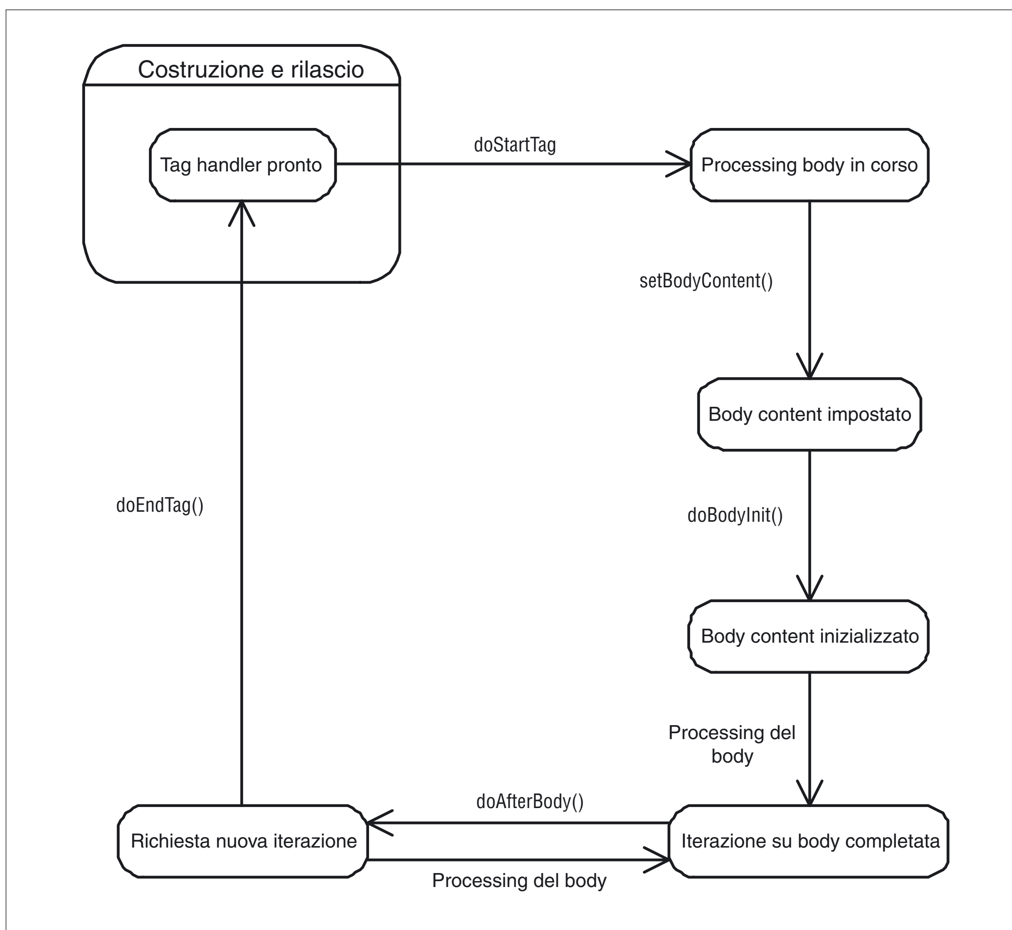
Il secondo elemento fondamentale di una Custom Tag Library è dato dal tag library descriptor — in breve TLD — ossia un documento XML avente suffisso .tld e contenente, in forma dichiarativa, le specifiche strutturali, sintattiche e implementative della CTL stessa.

Si noti che JSP 1.2 garantisce la piena backward compatibility con i tag library descriptor realizzati per la precedente versione 1.1 di JSP, il cui DTD continua ad essere pienamente supportato.

Una Custom Tag Library dichiara la propria intenzione di servirsi del nuovo Document Type Definition di JSP 1.2 semplicemente facendo ad esso riferimento nel suo tag <!DOCTYPE>.

Nel caso di CTL complesse il tag library descriptor tende a divenire un documento piuttosto lungo e prolisso, la cui stesura viene tipicamente automatizzata dagli wizard dell'application server ovvero dell'ambiente di sviluppo utilizzato. Per questa ragione non analizzeremo in detta-

Figura 6.3 – *Ciclo di vita di un tag handler di contenuto in modalità bufferizzata.*



glio tutti i numerosi tag supportati dal TLD Document Type Definition di JSP 1.2, ma ci limiteremo ad analizzarne i tre principali.

Il root element di un TLD è rappresentato dal tag <taglib>, la cui definizione legge:

```
<!ELEMENT taglib (tlib-version, jsp-version?, short-name, uri?, display-name?, small-icon?, large-icon?,
description?, validator?, listener*, tag+) >
```

I principali elementi di tale definizione sono:

- tlib-version: numero di versione della Custom Tag Library, gestito dal suo autore;

- jsp-version: versione opzionale del framework JSP per cui la CTL corrente è stata realizzata;

- short-name: identificatore associato alla tag library; tipicamente viene scelto e utilizzato a tal fine il prefisso con cui per convenzione si importa la Custom Tag Library nelle pagine JSP che ne fanno uso;

- description: descrizione libera e opzionale della tag library;

- validator: questo elemento opzionale rappresenta una novità di JSP 1.2 e permette di specificare un'apposita classe Java deputata alla validazione a livello globale delle pagine JSP che fanno uso della Custom Tag Library in oggetto;

- listener: tramite gli elementi opzionali listener, anch'essi introdotti a partire da JSP 1.2, è possibile stabilire un binding tra tag CTL e gli eventi globali dell'applicazione definiti a partire dalla versione 2.3 della Servlet API;

- tag: la definizione dei tag caratterizzanti una CTL viene fornita tramite uno o più elementi <tag> innestati nel root tag <taglib>.

L'elemento <tag> costituisce il singolo elemento di maggiore importanza di un TLD:

```
<!ELEMENT tag (name, tag-class, tei-class?, body-content?, display-name?, small-icon?, large-icon?,
description?, variable*, attribute*, example?) >
```

- name: identificatore del tag;

- tag-class: riferimento al tag handler che implementa il tag in oggetto, specificato attraverso il nome completo della classe Java che ne fornisce l'implementazione;

- tei-class: se presente fa riferimento ad una classe figlia di javax.servlet.jsp.tagext.TagExtraInfo, utilizzata — come avremo l'occasione di vedere — per definire variabili di scripting e implementare un meccanismo di validazione tag-based in fase di traduzione di pagina;

- body-content: questo elemento consente di specificare la natura del corpo del body del tag in oggetto e può assumere tre valori:

  - JSP: questo è il valore di default, e comunica al JSP container che il body content del tag deve venire processato a run time in maniera analoga al contenuto di ogni altro tag JSP e nel rispetto della sua sintassi;

  - tagdependent: il contenuto del tag non va interpretato in conformità agli standard JSP. Il JSP engine non ne effettuerà quindi nessuna elaborazione a runtime, ma si limiterà a trasmetterlo, integralmente e senza modifiche di sorta, al tag handler;

- ¤ empty: la presenza di un qualsiasi body content per il tag in oggetto è da considerarsi un errore e provocherà l'interruzione del processing della pagina JSP.

- variable: se presente, consiste in un elenco di variabili di scripting definite dal tag corrente e che verranno rese disponibili alle pagine JSP che ne fanno uso;

- attribute: sottoalbero opzionale composto di tag <attribute>, tramite i quali vengono specificati gli attributi accettati dal tag in oggetto;

- example: questo elemento opzionale costituisce una novità di JSP 1.2 e permette di fornire una descrizione libera — ed eventualmente comprensiva di esempi — della sintassi e della semantica del tag.

L'ultimo elemento su cui desideriamo soffermarci è il tag <attribute>, il quale dichiara le caratteristiche degli attributi accettati da un tag:

```
<!ELEMENT attribute (name, required?, rtexprvalue?) >
```

- name: identificatore dell'attributo;

- required: true se quest'attributo è obbligatorio, false in caso contrario; se non diversamente specificato ogni attributo è opzionale;

- rtexprvalue: questo elemento indica se il valore fornito per questo attributo va calcolato staticamente all'atto della compilazione della pagina o se esso può essere il risultato di un'espressione JSP da processare in sede di run time. La possibilità di valorizzare gli attributi in fase runtime — attivata impostando a true questo elemento — è evidentemente di fondamentale importanza e è su di essa che si basa la sincronizzazione dinamica tra il contenuto di pagine JSP e lo stato di oggetti JavaBean. D'altro canto l'impostazione di rtexprvalue a false — valore di default — rende disponibile agli eventuali strati di validazione tutte le informazioni circa l'utilizzo di un tag già in translation time, consentendone una completa analisi semantica.

Si noti che il valore di ogni attributo fornito ai tag delle Custom Tag Library deve essere delimitato da apici singoli o doppi, nel rispetto delle norme XML.

Utilizzo di una Custom Tag Library

L'intenzione di servirsi di una Custom Tag Library dall'interno di una pagina JSP va dichiarata tramite un'apposita direttiva <taglib>:

```
<%@ taglib uri="<uri>" prefix="<prefisso>" %>
```

L'attributo uri specifica la posizione del TLD associato alla Custom Tag Library, mentre l'attributo prefix comunica al motore JSP il prefisso con il quale si desidera fare riferimento ai tag della Custom Tag

Library nella pagina corrente. Si noti che il prefisso associato a una CTL non rappresenta una proprietà della tag library stessa, ma può essere liberamente scelto dall'autore della pagina JSP. Dal momento che ciò può rendere il codice confuso e difficilmente comprensibile, si consiglia di scegliere e utilizzare i prefissi delle CTL in maniera logica e coerente, attenendosi alla relativa documentazione. Uno standard comunemente accettato consiste nell'utilizzare prefissi chiari e significativi, facendo possibilmente uso al loro interno del nome della propria azienda o organizzazione — in maniera sostanzialmente analoga a quanto avviene per i package Java — al fine di evitare possibili conflitti nella nomenclatura delle Custom Tag Library. Vale qui la pena di notare come i prefissi jsp:, jspx:, java:, javax:, servlet:, sun: e sunw: siano da considerarsi riservati e non possano venire assegnati a tag library di terze parti.

Una pagina JSP può contenere un numero qualsiasi di direttive <taglib>, le quali possono essere collocate liberamente all'interno della pagina, purché antecedentemente al primo utilizzo di un tag della Custom Tag Library cui fanno riferimento. Per chiarezza si preferisce usualmente raggruppare tutte le direttive <taglib> utilizzate in testa alla pagina.

L'importazione di una Custom Tag Library il cui TLD risulti irraggiungibile in fase di traduzione ovvero l'utilizzo di un tag il cui tag handler non sia reperibile a run time fanno scattare delle apposite eccezioni e interrompono il processing della pagina JSP corrente.

Deploy di una Custom Tag Library

Il deploy di una Custom Tag Library può avvenire secondo tre diverse modalità. Il meccanismo consigliato, particolarmente adatto a Custom Tag Library di utilizzo generale per le quali si intende garantire il decoupling da una particolare applicazione o contesto web, consiste nel distribuire la CTL sotto forma di archivio JAR contenente le classi tag handler e il tag library descriptor, collocato nella directory /META-INF con il nome di taglib.tld.

A tale archivio JAR — tipicamente innestato nella directory /WEB-INF/lib/ delle web application per evitare problemi a livello di CLASSPATH — faranno quindi riferimento le direttive <taglib> delle diverse pagine JSP, secondo la seguente sintassi:

```
<%@ taglib uri="/WEB-INF/lib/<ctl.jar>" prefix="<prefisso>" %>
```

Nel caso di Custom Tag Library strettamente accoppiate a un'applicazione e non concepite in un'ottica di riusabilità è altresì possibile far uso di un meccanismo alternativo, consistente nell'innesto della CTL all'interno del file WAR della web application che ne fa uso. Per sfruttare questa tecnica è necessario utilizzare il tag <taglib> per inserire un riferimento alla Custom Tag Library nell'Application Deployment Descriptor dell'applicazione.

Il terzo e ultimo meccanismo — utilizzato nel contesto di sistemi e applicazioni particolarmente semplici o di test — consiste nel rendere il tag library descriptor di una Custom Tag Library accessibile tramite un URL pubblico e esterno alla web application, impostando al contempo il contesto di esecuzione in maniera che le relative classi tag handler si trovino nel CLASSPATH dell'applicazione stessa.

Validazione di una Custom Tag Library

Per validazione si intende la possibilità di stabilire dei vincoli di correttezza sintattica e semantica nell'utilizzo di una Custom Tag Library, il mancato rispetto dei quali porti ad intraprendere apposite azioni correttive o faccia scattare opportune procedure di errore.

Un primo livello di validazione è dato, in maniera abbastanza evidente, dallo stesso TLD, il quale è in grado di stabilire con una certa precisione i limiti sintattici dell'utilizzo di un tag CTL.

È d'altronde chiaro che a volte si rendono necessari dei meccanismi di verifica e controllo più flessibili, legati non solamente alla rigida definizione sintattica di un tag, ma al contesto semantico in cui esso viene utilizzato.

Il framework Java Server Pages offre due diversi meccanismi di custom validation, il primo dei quali è disponibile sin dalla versione 1.1 di JSP e consiste nell'istanziazione di una classe figlia di javax.servlet.jsp.tagext.TagExtraInfo, utilizzata per effettuare la verifica del corretto utilizzo di un tag in translation time. Tale meccanismo — per quanto di semplice implementazione — permette di definire politiche di validazione sufficientemente espressive per i bisogni della maggior parte delle Custom Tag Library.

Come abbiamo già visto in precedenza, l'associazione di un oggetto di classe TagExtraInfo a un tag di una Custom Tag Library viene dichiarata a livello di TLD tramite il subtag <tei-class> del tag <tag>, il quale contiene il nome completo della relativa classe Java, la quale verrà istanziata in sede di traduzione della pagina dal motore JSP.

L'implementazione di funzionalità di validazione tramite un oggetto TagExtraInfo richiede l'overriding di tre metodi:

```
public void setTagInfo(javax.servlet.jsp.tagext.TagInfo tagInfo)
```

L'accessore setTagInfo() viene invocato dal JSP engine sull'oggetto TagExtraInfo prima della validazione vera e propria. L'oggetto TagInfo, che normalmente le implementazioni di questo metodo si limitano a salvare in vista di un successivo utilizzo, contiene tutte le informazioni circa il tag corrente fornite nel tag library descriptor.

```
public javax.servlet.jsp.tagext.TagInfo TagInfo getTagInfo()
```

Il compito di getTagInfo() consiste semplicemente nel ritornare l'oggetto TagInfo precedentemente associato ad un oggetto di classe TagExtraInfo.

```
public boolean isValid(javax.servlet.jsp.tagext.TagData tagData)
```

isValid() implementa il vero e proprio meccanismo di validazione. Tale metodo viene invocato durante la fase di traduzione dal motore JSP per ogni utilizzo del tag all'interno di una pagina JSP, e il suo contratto consiste nel ritornare true qualora l'utilizzo del tag sia da ritenersi corretto, e false altrimenti. L'argomento tagData è sostanzialmente un data container per mezzo del quale isValid() può accedere alle informazioni circa gli attributi e i valori loro assegnati nel contesto corrente del tag. Si noti che, avvenendo la validazione in translation time, le informazioni di valorizzazione non risultano disponibili per gli attributi dinamici, ossia per quegli attributi il cui corrispondente elemento rtexprvalue nel tag library descriptor è stato impostato a true secondo quanto descritto in precedenza.

Al meccanismo di custom validation appena visto e basato sulla classe TagExtraInfo la nuova versione 1.2 di Java Server Pages affianca il concetto di validazione a livello di Custom Tag Library. In altre parole, si introduce la nozione di uno schema di validazione i cui confini semantici non siano limitati all'orizzonte del tag corrente, ma possano prendere in considerazione l'intera tag library e la sua dinamica di utilizzo all'interno di una singola pagina JSP.

A questo fine — come già visto in precedenza — è possibile dichiarare nel tag library descriptor di una Custom Tag Library una classe globale di validazione tramite l'attributo validator del root tag <taglib>. Essa deve estendere la classe astratta javax.servlet.jsp.tagext.TagLibraryValidator e viene invocata dal motore JSP tramite il seguente metodo di callback, definito dalle Servlet API 2.3:

```
javax.servlet.jsp.tagext.ValidationMessage[] validate(String prefix, String uri,
                                                      javax.servlet.jsp.tagext.PageData pageData)
```

Il JSP engine provvederà ad invocare il metodo validate() in fase di traduzione per ogni tag utilizzato, fornendo all'oggetto di classe TagLibraryValidator le seguenti informazioni:

- l'identificatore e l'URI del tag corrente, rispettivamente memorizzati nei parametri prefix e uri;
- un oggetto di classe javax.servlet.jsp.tagext.PageData contenente l'intera pagina JSP corrente sotto forma di documento XML.

Il metodo validate() è tenuto a ritornare null se l'utilizzo del tag è da ritenersi corretto e un vettore di messaggi di errore ValidationMessage in caso contrario.

Si noti che la precedente versione 2.2 delle Servlet API definisce il metodo validate() in maniera leggermente diversa, consentendo a quest'ultimo di ritornare un unico messaggio di errore codificato direttamente come oggetto String:

```
String validate(String prefix, String uri, javax.servlet.jsp.tagext.PageData pageData)
```



L'introduzione della classe TagLibraryValidator in JSP 1.2 rappresenta una novità particolarmente significativa e utile nell'ambito di Custom Tag Library ricche e complesse, in quanto consente a queste ultime di arrivare ad una decisione circa la validità del particolare utilizzo di un tag tenendo conto di tutti gli elementi della pagina (tag a qualsiasi livello di nesting, scriptlet, body content) e — se necessario — ritornando messaggi di errore chiari, comprensibili e localizzati.

Attributi

In maniera perfettamente analoga agli usuali tag HTML e XML, anche i tag definiti da una Custom Tag Library possono accettare uno più attributi, sia opzionali sia obbligatori, il cui fine consiste nella personalizzazione dell'effettivo comportamento di un tag.

L'implementazione di attributi in ambiente JSP avviene in maniera estremamente naturale: allo sviluppatore si richiede unicamente di implementare come property del relativo tag handler i tag definiti nel tag library descriptor.

Il JSP engine si farà quindi carico di tutta l'effettiva gestione dei tag, sfruttando a questo fine la Reflection API di Java. Vale la pena di notare come gli attributi possano essere basati su qualunque classe Java, e non solamente sui tipi primitivi. In effetti l'intera definizione degli attributi è totalmente type-agnostic, dal momento che nel TLD non viene specificata nessuna informazione a riguardo.

Il carattere di opzionalità di un attributo e la possibilità di essere valorizzato a run time anziché al momento della compilazione della pagina vengono decisi attraverso gli attributi required e rtexprvalue del tag <attribute> a livello di TLD.

È evidente come la parametrizzazione del comportamento di un tag sia ottenibile, oltre che tramite attributi, anche per mezzo di informazioni fornite al tag sotto forma di body content/tag nesting.

A titolo di esempio, consideriamo un'ipotetica CTL adabas — il cui fine consista nel fornire una possibilità di accesso non mediato ad una base dati ADABAS — e in particolare un tag <connect> il quale permetta, sulla base di un apposito insieme di informazioni di localizzazione e autenticazione, di stabilire una connessione verso il server. Appaiono possibili in linea di principio due definizioni del tag, la prima basata su attributi e la seconda basata su body content/tag nesting, alla luce delle quali le relative invocazioni avrebbero l'aspetto riportato di seguito.

Parametrizzazione del tag tramite attributi:

```
<adabas:connect server="adabas-srv.acme.net" user="XXX" password="YYY" />
```

Parametrizzazione del tag tramite body content/tag nesting:

```
<adabas:connect>
  <adabas:server>adabas-srv.acme.net</adabas:server>
  <adabas:user>XXX</adabas:user>
  <adabas:password>YYY</adabas:password>
</adabas:connect>
```

Entrambe queste possibilità sono perfettamente lecite e la questione di quale delle due sia da considerarsi preferibile o più corretta ha rappresentato negli ultimi anni un tema di acceso dibattito teorico, tuttora non completamente sopito.

Gli attributi presentano il vantaggio di una gestione innegabilmente semplice e intuitiva; d'altra parte le limitazioni sulla natura dei valori che possono essere così forniti a un tag sono estremamente significative. In particolare non è possibile fornire tramite attributi valori multipli, complessi o strutturati. Alla luce di queste considerazioni, ci sentiamo di proporre in ambito JSP le seguenti linee guida:

- far libero uso di attributi per comunicare a un tag valori semplici, essenzialmente numeri e stringhe, la cui complessità non è probabilmente destinata ad aumentare nel tempo;

- prediligere body content/tag nesting nel caso di valori complessi o strutturati, o per i quali non ci si voglia precludere nessuna possibilità di espansione futura;

- a un livello più generale, consigliamo l'utilizzo di tecniche basate su body content e tag nesting per rappresentare valori semanticamente legati al contenuto del tag, piuttosto che alla natura delle operazioni richieste al tag per fare fronte ai propri compiti; in altre parole reputiamo corretto utilizzare gli attributi per descrivere il "come" di un tag e il body content/tag nesting per descriverne il "cosa".

Qualora si desideri ricorrere al body content/tag nesting per rimpiazzare un attributo per il quale sia stata configurata a livello di TLD l'impostazione rtexprvalue sarà necessaria – come avremo modo di vedere in seguito — la definizione da parte del tag di un apposito set di variabili di scripting richiamabili dall'interno del body content.

Si noti che le linee guida proposte, notevolmente meno restrittive nei confronti degli attributi di quanto oggi consigliato da gran parte della comunità XML e sostanzialmente in linea con la visione delineata da Sun nel suo J2EE Tutorial, sono esplicitamente pensate per l'implementazione di Custom Tag Library in ambito JSP, e non sono necessariamente valide in contesti più generali — quali la produzione o l'elaborazione di documenti XML in situazioni e con framework differenti — all'interno dei quali alcune delle assunzioni su cui le nostre linee guida sono basate potrebbero non rivelarsi più corrette. In linea generale, ad esempio, ogni utilizzo di attributi all'interno di documenti XML validati su base DTD conduce a delle potenziali incongruenze, non essendone possibile in sede di Document Type Definition una soddisfacente verifica semantica, questione al contrario elegantemente risolta dal paradigma Java Server Pages attraverso quei meccanismi di custom validation che abbiamo già avuto modo di considerare e apprezzare.

Variabili di scripting

Nella nostra trattazione dei tag abbiamo sino ad ora sempre sottinteso che il fine di un tag consista, in ultima analisi, nella produzione di output a partire da un determinato insieme di dati di input, specificati sotto forma di attributi o di body content/tag nesting; in quest'ultimo caso abbiamo menzionato la possibilità di procedere in maniera ricorsiva, innestando nel body content ulteriori tag, in numero e con profondità a piacere, sia standard JSP — inclusi gli oramai sconsigliati scriptlet — sia provenienti dalla stessa o da altre Custom Tag Library.

È tuttavia immaginabile un'ampia casistica di tag il cui obiettivo non sia tanto quello di produrre output quanto piuttosto di fornire informazioni a un altro strato di tag, sulla scorta delle quali procedere all'effettiva generazione di contenuti. Un classico esempio è rappresentato da ogni contesto nel quale si desideri dividere il ruolo del reperimento o produzione delle informazioni dal layer di decodifica e presentazione, tipicamente basato su linguaggi HTML o XML / XSL.

Tale funzionalità viene implementata in ambito JSP dalle cosiddette variabili di scripting, tramite le quali diversi tag possono comunicare e scambiarsi informazioni a vicenda.

Le variabili di scripting possono essere definite secondo due diverse modalità:

- dichiarativa:, introdotta a partire dalla versione 1.2 di JSP, essa è composta di una definizione di interfaccia a livello di TLD e di una implementazione in sede di tag handler;

- comportamentale: presente sin dalla versione 1.1 di JSP, essa si differenzia dalla modalità dichiarativa per il fatto che tanto la dichiarazione quanto l'impostazione delle variabili avvengono a livello di codice Java.

Si noti che le suddette modalità di definizione sono mutuamente esclusive: ogni tag che definisca delle variabili di scripting deve farlo o in modalità integralmente dichiarativa o in modalità integralmente comportamentale.

La modalità dichiarativa è caratterizzata da una notevole semplicità di utilizzo ed è da considerarsi come lo schema standard di definizione di variabili di scripting da utilizzare in tutte le nuove Custom Tag Library. Essa passa per la definizione delle variabili a livello di TLD attraverso l'utilizzo del tag <variable>, figlio dell'elemento <tag> analizzato in precedenza:

```
<!ELEMENT variable ((name-given | name-from-attribute), variable-class?, declare?, scope?) >
```

I principali componenti di tale tag sono:

- name-given: deve essere un identificatore valido secondo le regole di sintassi Java e rappresenta il nome con cui la variabile di scripting risulterà accessibile in contesto JSP;

- name-from-attribute: rappresenta una possibilità alternativa a name-given per specificare il nome di una variabile di scripting, il quale verrà desunto dal valore dell'attributo specificato del tag corrente in sede di compilazione di pagina; si noti che per tale attributo non sarà quindi possibile impostare il flag rtexprvalue;

- variable-class: nome completo della classe Java che costituisce il tipo della variabile; se non specificato esso vale per default java.lang.String;

- declare: parametro booleano impostabile a true / yes ovvero false / no, il quale determina se la variabile di scripting sia da intendersi alla stregua di una nuova variabile o se costituisca invece una ridefinizione di una variabile già esistente e inserita nell'oggetto javax.servlet.jsp.PageContext della pagina JSP da un tag processato in precedenza. Di norma si raccomanda di impostare ovvero di lasciare declare al suo valore di default pari a true. Così facendo in caso di doppia definizione di variabili con scope contrastanti da parte di due tag separati o innestati, il motore JSP sarà in grado di produrre un opportuno messaggio d'errore in translation time, prevenendo quindi conflitti e bug di difficile identificazione e soluzione;

- scope: JSP supporta tre classi di scope per le variabili di scripting, e precisamente:

  - javax.servlet.jsp.TagExt.VariableInfo.NESTED: la variabile risulterà accessibile nel body content a partire dal tag di apertura sino al relativo tag di chiusura, e eventualmente anche a tutti i tag innestati. A livello del tag handler padre essa potrà quindi venire utilizzata e impostata all'interno dei metodi doStartTag(), doInitBody() e doAfterBody(). Il livello di

scope NESTED minimizza i rischi di conflitti con variabili definite da altri tag e è pertanto la classe di scope raccomandata nel normale sviluppo di librerie CTL e rappresenta il valore di default per il parametro scope.

- javax.servlet.jsp.TagExt.VariableInfo.AT_BEGIN: lo scope di validità della variabile parte dal tag di apertura e si protrae sino al termine dell'intera pagina JSP corrente. Il tag handler padre potrà quindi farne uso in tutti i metodi, vale a dire doStartTag(), doInitBody(), doAfterBody() e doEndTag();

- javax.servlet.jsp.TagExt.VariableInfo.AT_END: lo scope di validità della variabile parte dal tag di chiusura e si protrae sino al termine dell'intera pagina JSP corrente. Il tag handler padre potrà quindi farne uso esclusivamente nel metodo doEndTag().

Il secondo passo nella definizione di una variabile di scripting in modalità dichiarativa consiste nella sua impostazione a livello di tag handler da parte di uno o più dei metodi autorizzati in tal senso in funzione della classe di scope dichiarata. La variabile non sarà effettivamente disponibile alla pagina JSP, vale a dire al body content o ad eventuali tag innestati o successivi, sino alla sua effettiva impostazione. Le variabili di scripting vengono trattate come attributi dell'oggetto PageContext corrente e vengono quindi impostate tramite una semplice invocazione del relativo metodo setAttribute(); questo significa che variabili il cui valore sia costituito da un tipo primitivo dovranno essere memorizzate tramite la rispettiva wrapping class: java.lang.Integer per variabili di tipo int, e così via.

Vale qui la pena di porre l'accento sulla semplicità del metodo dichiarativo di definizione di variabili di scripting, il quale nella maggior parte dei casi non consiste in altro che nell'aggiunta di tre o quattro semplici righe al tag library descriptor e nella scrittura di una o due righe di codice nel tag handler. Eppure, a fronte di queste semplici operazioni, il risultato consiste nell'implementazione di una primitiva di comunicazione tra tag potente e flessibile, sulla scorta della quale è effettivamente possibile garantire quella netta separazione di ruoli e responsabilità che è presupposto fondamentale alla realizzazione di tag e CTL riusabili e adattabili ad un ampio spettro di situazioni.

Il secondo schema di definizione di variabili di scripting consiste nella cosiddetta modalità comportamentale, caratteristica della versione 1.1 del framework Java Server Pages; come avremo modo di vedere, essa richiede allo sviluppatore uno sforzo implementativo lievemente maggiore e il suo uso viene oggi limitato a poche e ben definite circostanze.

La modalità comportamentale di definizione della variabili si articola in tre passi fondamentali:

- dichiarazione di una classe figlia di javax.servlet.jsp.tagext.TagExtraInfo a livello di tag library descriptor;

- implementazione della suddetta classe TagExtraInfo;

- impostazione delle variabili di scripting a livello di tag handler in maniera esattamente identica a quanto avviene nella modalità dichiarativa.

La dichiarazione della classe TagExtraInfo viene effettuata attraverso l'elemento <tag> del TLD. È da notare come, in questo caso, non vada definita nessuna variabile di scripting tramite il sottotag <variable> dell'elemento <tag>.

Il secondo passo consiste nell'implementazione di una classe figlia di javax.servlet.jsp.tagext.TagExtraInfo, la quale possiede in ambito JSP 1.2 due diverse funzioni: con la prima abbiamo già fatto conoscenza allorché abbiamo utilizzato tale classe al fine di implementare dei meccanismi di custom validation. Il secondo ruolo ricoperto dalla classe TagExtraInfo consiste nella dichiarazione di un insieme di variabili di scripting per tramite del seguente metodo:

```
public javax.servlet.jsp.tagext.VariableInfo[] getVariableInfo(javax.servlet.jsp.tagext.TagData tagData)
```

La responsabilità del metodo getVariableInfo() consiste nel ritornare al motore JSP un vettore di oggetti VariableInfo, ognuno dei quali definisce una variabile di scripting in maniera esattamente analoga a quanto possibile con il tag <variable> nella schema dichiarativo di definizione delle variabili di scripting. Per convincerci di questo sarà sufficiente considerare le property della non-mutable class VariableInfo:

- java.lang.String varName: essa rappresenta, in maniera identica al parametro name-given del tag <variable>, il nome della variabile di scripting;

- java.lang.String.className: nome completo della backend-class della variabile di scripting, analogo al parametro variable-class;

- boolean declare: richiesta di definizione di una nuova variabile alla medesima stregua del parametro declare;

- int scope: classe di scope della variabile di scripting corrispondente al parametro scope del tag <variable> e impostabile ai medesimi valori (NESTED, AT_BEGIN e AT_END).

Una tipica implementazione del metodo getVariableInfo() consisterà quindi semplicemente nella ripetuta istanziazione di oggetti VariableInfo(), nella loro coagulazione in un vettore e nel loro ritorno al JSP engine.

Da quanto detto emerge con chiarezza la caratteristica precipua del metodo comportamentale di definizione di variabili di scripting: la dichiarazione a livello di codice Java permette infatti allo sviluppatore di definire in maniera dinamica il numero e le caratteristiche delle variabili di scripting definite da un tag — eventualmente in dipendenza dal contesto e da altri parametri ambientali — laddove il metodo dichiarativo comporta una dichiarazione statica e immutabile delle variabili di scripting.

L'utilizzo della modalità comportamentale di definizione della variabili di scripting — cui si possono in ogni caso muovere fondate obiezioni in ragione dei problemi di chiarezza e manutenibilità del codice e della pagine JSP cui essa conduce — è raccomandato solamente in quegli infrequenti casi in cui si manifesti la reale esigenza di disporre di un insieme di variabili di scripting di cui non siano noti a priori il numero e il tipo.

Come abbiamo già fatto notare in precedenza, non è possibile per un medesimo tag definire variabili di scripting sia in modalità dichiarativa sia comportamentale. Questo significa in pratica che, qualora un tag definisca le proprie variabili di scripting a livello di TLD tramite il tag <variable>, il metodo getVariableInfo() del suo eventuale oggetto TagExtraInfo dovrà ritornare null. Per converso, qualora tale metodo non ritorni null, non sarà ammissibile per il tag in questione far uso del tag <variable> nel proprio tag library descriptor.

Tag nesting

Le API JSP permettono a uno o più tag di riconoscere il proprio contesto di nesting e di comportarsi di conseguenza in maniera "intelligente", vale a dire cooperando con il proprio parent tag al perseguimento di un obiettivo comune. Abbiamo già presentato un esempio implicito di tag innestati in sede di trattazione degli attributi, allorché abbiamo mostrato un possibile esempio di codice JSP:

```
<adabas:connect>
  <adabas:server>adabas-srv.acme.net</adabas:server>
  <adabas:user>XXX</adabas:user>
  <adabas:password>YYY</adabas:password>
</adabas:connect>
```

Quello che ci aspetta a fronte del suddetto codice è che i tre tag innestati <adabas:server>, <adabas:user> e <adabas:password> cooperino con il proprio parent tag <adabas:connect>, comunicandogli lo hostname del server e i relativi parametri di autenticazione, e permettendogli quindi — presumibilmente nel metodo doEndTag() del relativo tag handler — di stabilire l'effettiva connessione al server.

Il framework Java Server Pages permette in effetti di realizzare tag dotati di questo genere di "intelligenza" e il nostro obiettivo è esporre gli strumenti forniti allo sviluppatore in questo senso e comprenderne le funzionalità e i possibili schemi di utilizzo.

L'idea che un tag sia in grado di cooperare con il proprio contesto, rintracciando i tag con cui è in grado di cooperare, o per conoscenza diretta o sulla scorta di una comune interfaccia funzionale, è per sua natura essenzialmente dinamica.

Per questo motivo il tag nesting è sostanzialmente incompatibile con una dichiarazione a livello di TLD, che sarebbe forzatamente statica — vale qui lo stesso principio analizzato nel paragrafo precedente in relazione alla definizione delle variabili di scripting — e viene piuttosto definito e implementato interamente a livello di codice Java.

Sarà d'altronde il caso di notare come l'impossibilità di fornire una descrizione formale del meccanismo di tag nesting a livello di XML non permetta al motore JSP l'implementazione automatica di meccanismi di validazione preventiva o cooperativa, delegando di fatto tale onere allo sviluppatore della Custom Tag Library.

Effettuate queste considerazioni generali possiamo passare all'analisi delle API che implementano il tag nesting. Come più volte ci è successo nel corso della nostra cavalcata all'interno del mondo JSP/JSTL, non possiamo fare a meno di stupirci di fronte alla semplicità dei meccanismi che soggiacciono all'implementazione di una funzionalità che, di primo acchito, potrebbe non sembrare esattamente elementare.

In effetti l'intera struttura del tag nesting si basa su di un unico metodo statico della classe javax.servlet.jsp.tagext.TagSupport:

```
public static final javax.servlet.jsp.tagext.Tag findAncestorWithClass(javax.servlet.jsp.tagext.Tag from,
                                                                        java.lang.Class class)
```

Tale metodo cerca, nella struttura gerarchica dei tag della pagina JSP corrente, il primo tag posto in un livello gerarchico superiore al tag rappresentato dal parametro from e il cui tag handler sia implementato da una classe di tipo class.

Esso permette quindi a un tag handler di rintracciare, se esiste, un'istanza di tag con cui esso sia in grado di cooperare e di porsi con essa in diretta comunicazione.

Si noti che tale metodo permette di trovare il tag handler desiderato indipendentemente dal livello di nesting e rappresenta quindi un meccanismo assai più generale e flessibile rispetto all'invocazione, eventualmente ripetuta, del metodo getParent(), per cui tramite un tag handler ottiene un riferimento al parent tag, vale dire al tag handler di livello gerarchico immediatamente superiore all'interno della pagina JSP corrente.

Torniamo ora al nostro esempio, supponendo che il tag handler del tag <adabas:connect> sia rappresentato dalla classe AdabasTagHandler la quale, tra le altre, possiede una proprietà serverName pari al nome host del server verso il quale deve avvenire la connessione. Il tag handler del tag <adabas:server> potrebbe leggere e comunicare il hostname del server al tag <adabas:connect> attraverso la seguente, elementare definizione del metodo doEndTag(), dalla quale omettiamo, per brevità, ogni gestione di errori e eccezioni:

```
public int doEndTag() throws JspTagException {
  BodyContent bodyContent=getBodyContent();
  String hostName=bodyContent.getString();
  AdabasTagHandler adabasTH
    =(AdabasTagHandler) TagSupport.findAncestorWithClass(this,AdabasTagHandler.class();
  adabasTH.setServerName(hostName);
}
```

Eccezioni

I meccanismi di validazione formale offerti dal framework Java Server Pages attraverso il tag library descriptor e gli oggetti custom validator permettono di prevenire in translation time gran parte degli errori e delle inconsistenze che si possono verificare in fase di creazione e utilizzo di Custom Tag Library e di applicazioni JSP.

È tuttavia ragionevole attendersi che diverse condizioni di errore si manifesteranno unicamente in sede di run time, ponendo al tag handler il problema di come reagire in tali situazioni. Sostanzialmente sono identificabili due strategie di gestione degli errori.

Nel primo caso, si è presentata una condizione di errore critico a fronte del quale è impossibile o indesiderabile per il tag procedere nella propria elaborazione; se il tag svolge un ruolo centrale all'interno della pagina JSP corrente potrebbe per quest'ultima essere inattuabile la produzione

di un output corretto o in linea con le attese dell'utente. In questo caso la soluzione consiste usualmente nell'interrompere l'intero processing della pagina: il tag handler solleva un'eccezione javax.servlet.jsp.JspTagException, la quale porta alla generazione di una pagina di errore eventualmente personalizzata.

Nel seconco caso, si è manifestata una condizione di errore secondario, la quale non preclude in linea di massima la generazione di una pagina corretta; il tag produrrà tipicamente un output nullo o contenente una segnalazione di errore e imposterà eventualmente degli appositi flag per mezzo di opportune variabili di scripting, delegando la scelta definitiva sul da farsi a dei tag di livello superiore o direttamente alla logica principale di controllo della pagina JSP corrente.

Le API 1.2 del framework JSP introducono un nuovo, terzo schema di gestione degli errori, basato sull'interfaccia javax.servlet.jsp.tagext.TryCatchFinally, implementando la quale ogni tag handler può reagire ad un'eventuale eccezione sollevata da un altro tag handler. Tale interfaccia consta di due metodi:

```
public void doCatch(java.lang.Throwable t)

public void doFinally()
```

doCatch() viene invocato dal motore JSP se un qualsiasi tag handler solleva un'eccezione — sia checked che unchecked — durante il processing di un body content, ovvero dall'interno dei metodi doStartTag(), doEndTag(), doAfterBody() e doInitBody(). Si noti che doCatch() non viene invocato qualora l'eccezione sia stata sollevata dai metodi setParent(), setPageContext(), setBodyContent() o release().

Il tag handler può, all'interno del metodo doCatch(), gestire l'eccezione e impedire così che essa raggiunga i tag handler di livello superiore ed eventualmente la logica JSP della pagina corrente. In alternativa esso può rigenerare la medesima o un'altra eccezione, propagandola così al livello superiore.

In ogni caso, anche qualora doCatch() risolva il problema segnalato dall'eccezione sollevata, il motore JSP non invocherà ulteriori metodi di callback sul tag corrente.

Il senso del metodo doFinally() — come si può evincere dal nome — consiste nel permettere a ogni tag handler, anche in presenza di un'eccezione, di rilasciare in modo corretto tutte le risorse acquisite nel corso della propria elaborazione.

Se presente, il metodo doFinally() viene sempre invocato al termine dell'attività di un tag handler, sia che essa avvenga normalmente — in questo caso doFinally() verrà attivato successivamente al metodo doEndTag() — sia che essa avvenga per causa di una situazione di errore non gestita.

L'interfaccia TryCatchFinally, che per motivi di completezza non abbiamo potuto esimerci dal trattare, va considerata alla sorta di una vera e propria extrema ratio, cui ricorrere in situazioni rare e ben definite.

Tale interfaccia infatti, oltre a duplicare in maniera inelegante il meccanismo delle eccezioni già presente nel linguaggio Java, sottintende l'idea che i tag handler possano utilizzare durante il loro intero ciclo di vita connessioni a risorse critiche e di cui quindi vada necessariamente garantito il rilascio in caso di interruzione del processing della pagina JSP, quali in prima istanza connessioni a banche dati.

Sarà qui il caso di chiarire una volta di più come tale concezione — allineata a quello che molti sviluppatori considerano essere il fine dei componenti DB della libreria JSTL — non sia da considerarsi valida in generale.

È certamente lecito, come da noi già fatto notare, utilizzare a tale fine il framework JSTL in ambienti di prototipazione o per la realizzazione di applicazioni a basso budget, ma tale schema di utilizzo non può essere esteso a regola generale, nella misura in cui tende a caricare gli strati di interfaccia utente JSP e JSTL dei compiti propri dei layer di business logic quali Enterprise Java Beans e Java Data Objects.

Eventi

L'ultimo componente di una Custom Tag Library sul quale vogliamo soffermarci rappresenta una integrale novità introdotta a partire dalla versione 2.3 delle Servlet API, intimamente legate alla versione 1.2 del framework Java Server Pages. Stiamo parlando degli eventi a livello di applicazione, comunemente detti *application lifecycle event*. In maniera esattamente analoga alle normali applicazioni Swing, è ora possibile per un tag handler qualificarsi come listener e ricevere notifica di uno o più eventi.

Gli eventi generati dalle Servlet API 2.3 vengono suddivisi in due principali categorie: gli eventi legati al servlet context e gli eventi di tipo HTTP session.

Il trigger della prima classe di eventi, rappresentati dalla classe javax.servlet.ServletContextEvent e dalle sue sottoclassi, avviene contestualmente alle transizioni di stato del servlet context associato alla web application corrente. A questo fine vengono definite due interfacce: javax.servlet.ServletContextListener e javax.servlet.ServletContextAttributeListener.

Tramite javax.servlet.ServletContextListener, un oggetto listener riceve notifica dei seguenti eventi globali a livello di servlet context:

- public void contextInitialized(javax.servlet.ServletContextEvent sce): notifica della creazione del servlet context e dell'attivazione della web application corrente;
- public void contextDestroyed(javax.servlet.ServletContextEvent sce): notifica della prossima distruzione del servlet context corrente e della terminazione della relativa web application.

javax.servlet.ServletContextAttributeListener invece è deputata a notificare agli oggetti listener ogni modifica intercorsa alla tabella di attributi del servlet context corrente tramite i seguenti eventi (si noti che tutti i seguenti metodi vengono invocati a modifica degli attributi già avvenuta):

- public void attributeAdded(javax.servlet.ServletContextAttributeEvent scae): un nuovo attributo, i cui dati sono disponibili come proprietà dell'oggetto ServletContextAttributeEvent passato al gestore di eventi, è stato aggiunto alla tabella di attributi del servlet context corrente;
- public void attributeRemoved(javax.servlet.ServletContextAttributeEvent scae): un attributo, i cui parametri sono ancora accessibili tramite il parametro di evento, è stato rimosso dalla tabella di attributi del servlet context corrente;

- public void attributeReplaced(javax.servlet.ServletContextAttributeEvent scae): un attributo è stato modificato e il suo precedente valore è accessibile tramite l'oggetto ServletContextAttributeEvent.

Le nuove JSP API permettono tanto a ogni web application quanto a ogni Custom Tag Library di registrarsi come listener per i suddetti eventi; questa funzionalità può effettivamente rivelarsi estremamente utile in tutti quei casi in cui una CTL si trovi a dover gestire una risorsa condivisa non a livello di applicazione, ma a livello globale di tag library.

Laddove sino ad ora l'unica possibilità sarebbe consistita nel far acquisire e rilasciare la risorsa in ogni pagina JSP dal singolo tag handler — soluzione non corretta da un punto di vista teorico e fonte di probabili problemi a livello di performance — la versione 1.2 del framework Java Server Pages, combinata alle Servlet API 2.3, permette di risolvere il problema in maniera semplice e elegante, implementando all'interno della Custom Tag Library in oggetto un ServletContextListener, il quale provveda ad acquisire e rilasciare la risorsa condivisa rispettivamente all'attivazione e alla distruzione del servlet context associato all'applicazione web.

All'implementazione di un listener a livello di sorgente Java deve corrispondere una sua dichiarazione all'interno del tag library descriptor, la quale assume la forma di un tag <listener> da inserirsi all'interno del tag <taglib> della Custom Tag Library, prima degli elementi <tag> e successivamente all'eventuale elemento opzionale <validator>.

La seconda classe di eventi è costituita dagli eventi di tipo HTTP session — implementati tramite istanze della classe javax.servlet.http.HttpSessionEvent o di sue sottoclassi — i quali scattano in funzione delle fasi del ciclo di vita dell'oggetto HttpSession corrente, tramite il quale risulta possibile identificare e tracciare gli utenti di un'applicazione web; ancora una volta ci troviamo di fronte a due interfacce, perfettamente parallele alla ServletContextListener e alla ServletContextAttributeListener esaminate in precedenza.

Analogamente a quanto discusso per i servlet context event, anche nel caso di eventi di tipo HTTP session è possibile per web application e Custom Tag Library operanti in contesti JSP 1.2 registrarsi come listener. Va comunque notato che l'utilità pratica legata all'intercettazione degli HttpSessionEvent tende ad assumere connotazioni piuttosto verticalizzate, tipicamente legate all'esecuzione di "meta-azioni' a livello globale di web application o di application server.

Gestori di eventi HTTP session vengono, a titolo di esempio, utilizzati per implementare sistemi di monitoraggio delle performance, di statistiche del traffico e della frequenza degli accessi e per l'implementazione e l'enforcing di politiche di licensing basate su tetti massimi di utenti.

Le novità di JSP 2.0 e della Servlet API 2.4

Con l'argomento Custom Tag Library si chiude questo capitolo, incentrato su framework e tool rivolti all'implementazione di Web User Interface in applicazioni Java nel contesto della versione 1.2 di Java Server Pages e della versione 2.3 delle Servlet API.

Riteniamo tuttavia sensato non trascurare le novità in corso di maturazione su questo fronte e intendiamo quindi analizzare brevemente le principali proposte in cantiere per la release 2.4 delle Servlet API e per la versione 2.0 del framework Java Server Pages, in sincronia con la versione 1.4 delle specifiche J2EE.

Nel caso delle Servlet API è da notare come la solidità oramai raggiunta da questo prodotto traspaia dalle richieste di modifiche contenute nel rispettivo documento di specifiche, tutte in prima istanza votate al consolidamento e all'evoluzione delle strutture esistenti.

Presentano particolare interesse le seguenti proposte di modifica:

- le Servlet API 2.4 richiedono servlet container allineati alla versione 1.1 del protocollo HTTP e alla versione 1.3 di J2SE;

- in linea con le recenti evoluzioni nel mondo J2EE la definizione del deployment descriptor non viene più affidata ad un DTD ma a un documento XML Schema, la cui superiore ricchezza ed espressività semantica garantisce una sostanziale maggiore espandibilità e al contempo più complete funzionalità di verifica e di controllo; i deployment descriptor caratteristici delle versioni 2.2 e 2.3 della Servlet API rimangono comunque pienamente supportati per ragioni di backwards compatibility;

- viene definita una nuova classe di eventi a livello di applicazione, rappresentati dalle interfacce ServletRequestEvent e ServletRequestAttributeEvent nonché dalle rispettive interfacce listener, tramite le quali un'applicazione o una Custom Tag Library può reagire alle richieste HTTP processate dal servlet container;

- l'interfaccia ServletRequest dispone dei nuovi metodi getLocalAddr(), getLocalName(), getLocalPort() e getRemotePort() rivolti all'acquisizione di informazioni circa le connessioni lato client e server;

- un'importante attività di razionalizzazione e riordino è stata eseguita sui request dispatcher, i quali beneficiano inoltre di alcune nuove e significative funzionalità; in particolare il nuovo tag <dispatcher> definito a livello di deployment descriptor permette di attivare l'esecuzione di uno o più filtri in seguito all'invocazione dei metodi forward() o include();

- i meccanismi di internazionalizzazione e il supporto di charset differenti sono stati oggetto di importanti chiarificazioni e espansioni, sia a livello di API attraverso le interfacce ServletResponse e ServletResponseWrapper sia per quanto riguarda l'introduzione del nuovo elemento <locale-encoding-mapping-list> nel deployment descriptor;

- l'interfaccia SingleThreadModel, utilizzata per impedire che un servlet container richiami da più thread concorrenti il metodo service() di un servlet programmato in maniera non rientrante, è posta in stato deprecated;

- viene reso ufficiale e obbligatorio il supporto — peraltro già presente in diversi application server — di welcome file generati in maniera dinamica.

Le novità introdotte nell'attuale bozza di specifiche della versione 2.0 del framework Java Server Pages sono invece ben più corpose, per quanto già largamente annunciate:

- le specifiche JSP 2.0 richiedono la piattaforma J2SE 1.4 all'interno di contesti J2EE versione 1.4; viceversa, nel caso di servlet container stand-alone, è richiesta la versione 1.3 di J2SE;
- prerequisito del framework JSP 2.0 è la versione 2.4 delle Servlet API;
- la libreria JSTL (JSP Standard Tag Library) e il linguaggio EL (Expression Language) da noi trattati in maniera approfondita nel corso del presente capitolo sono ora parte integrante di Java Server Pages;
- stretta integrazione con il framework Java Server Faces, anch'esso oggetto della nostra attenzione e destinato a diventare lo standard di riferimento per la realizzazione di Web User Interface in Java;
- viene introdotta un'API semplificata per la definizione di tag handler e basata sull'estensione della classe `javax.servlet.jsp.tagext.SimpleTagSupport` e sull'override del solo metodo `public void doTag()`;
- è contemplata la possibilità, destinata a personale privo di specifici skill di programmazione, di realizzare semplici custom tag facendo uso esclusivo della sintassi JSP;
- il ruolo e le potenzialità del formalismo XML di documenti JSP sono stati significativamente estesi e potenziati;
- espressioni in linguaggio EL possono essere liberamente utilizzate all'interno di pagine JSP senza dover necessariamente ricorrere al tag `<c:out>`;
- la classe `JspTagException` è stata dotata di costruttori allineati alla classe padre `JspException`, i quali rendono possibile sollevare eccezioni dotate di informazioni significative di contesto senza dover ricorrere a manipolazioni dello stack trace.

Bibliografia

[JSRV22] "JSR-903 – Java Servlet 2.2 Specification"
http://www.jcp.org/aboutJava/communityprocess/maintenance/jsr903/

[JSP11] "JSR-906 – Java Server Pages 1.1 Specification"
http://www.jcp.org/aboutJava/communityprocess/maintenance/jsr906/

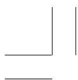
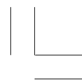
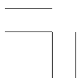
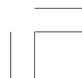
[JSP12] "JSR-53 – Java Servlet 2.3 and Java Server Pages 1.2 Specification"
http://www.jcp.org/aboutJava/communityprocess/review/jsr053/

[JSTL] "JSR-52 – JSP Standard Tag Library Specification"
http://jcp.org/aboutJava/communityprocess/final/jsr052/

[JSRV24] "JSR-154 – Java Servlet 2.4 Specification"
http://jcp.org/aboutJava/communityprocess/first/jsr154/

[JSP20] – "JSR-152 – Java Server Pages 2.0 Specification"
http://jcp.org/aboutJava/communityprocess/first/jsr152/

[JSF] "JSR-127 – Java Server Faces" http://www.jcp.org/en/jsr/detail?id=127

Capitolo 7

Java e CORBA

Gianluca Morello

Introduzione

Nell'era di Internet e delle grandi Intranet aziendali, il modello computazionale dominante è chiaramente quello distribuito. Comunemente in un ambiente distribuito convivono Mainframe, server Unix e macchine Windows; questo pone seri problemi di interoperabilità tra piattaforme differenti, sistemi operativi differenti e linguaggi differenti.

Lo scopo dichiarato delle specifiche CORBA è proprio quello di definire un'infrastruttura standard per la comunicazione tra e con oggetti remoti ovunque distribuiti, indipendentemente dal linguaggio usato per implementarli e dalla piattaforma di esecuzione. È bene quindi notare che, a differenza di altre tecnologie distribuite quali RMI, Servlet o EJB, non si sta parlando di una tecnologia legata ad una specifica piattaforma, ma di uno standard indipendente dal linguaggio adottato che può consentire ad oggetti Java di comunicare con "oggetti" sviluppati in COBOL, C++ o altri linguaggi ancora.

L'acronimo CORBA sta per *Common Object Request Broker Architecture* e non rappresenta uno specifico prodotto, bensì un'insieme di specifiche che definiscono un'architettura completa e standardizzata di cui esistono varie implementazioni. Le specifiche sono prodotte da OMG, un consorzio che comprende più di 800 aziende ed include i più illustri marchi dell'industria informatica.

L'elemento fondamentale dell'intera architettura è il canale di comunicazione degli oggetti nell'ambiente distribuito, l'*Object Request Broker* (ORB).

Le specifiche CORBA 1.1 furono pubblicate da OMG nell'autunno del 1991 e definivano un'API e un linguaggio descrittivo per la definizione delle interfacce degli oggetti CORBA detto *Interface Definition Language* (IDL). Soltanto nel dicembre del 1994, con CORBA 2.0, vennero definiti i dettagli sulla comunicazione tra differenti implementazioni di ORB con l'introduzione dei protocolli GIOP e IIOP.

Sebbene CORBA possa essere utilizzato con la maggior parte dei linguaggi di programmazione, Java risulta il linguaggio privilegiato per implementare le sue specifiche in un ambiente eterogeneo in quanto permette agli oggetti CORBA di essere eseguiti indifferentemente su mainframe, network computer o telefono cellulare. Nello sviluppo di applicazioni distribuite, Java e CORBA si completano a vicenda: CORBA affronta e risolve il problema della trasparenza della rete, Java quello della trasparenza dell'implementazione rispetto alla piattaforma di esecuzione.

Object Management Group

L'OMG è un consorzio no-profit interamente dedicato alla produzione di specifiche e di standard; vede la partecipazione sia dei grandi colossi dell'informatica, sia di compagnie medie e piccole.

L'attività di OMG cominciò nel 1989 con soli otto membri tra cui Sun Microsystems, Philips, Hewlett-Packard e 3Com. Le specifiche più conosciute prodotte dal consorzio sono sicuramente UML e CORBA che comunque nella logica OMG, rappresentano strumenti strettamente cooperanti nella realizzazione di applicazioni Enterprise OO.

Sin da principio il suo scopo è stato quello di produrre e mantenere una suite di specifiche di supporto per lo sviluppo di software distribuito in ambienti distribuiti, coprendo l'intero ciclo di vita di un progetto: analisi, design, sviluppo, runtime e manutenzione.

Per ridurre complessità e costi di realizzazione, il consorzio ha introdotto un framework per la realizzazione di applicazioni distribuite. Questo framework prende il nome di *Object Management Architecture* (OMA) ed è il centro di tutte le attività del consorzio; all'interno di OMA convivono tutte le tecnologie OMG.

Nell'ottica OMG la definizione di un framework prescinde dall'implementazione e si "limita" alla definizione dettagliata delle interfacce di tutti i componenti individuati in OMA. I componenti di OMA sono riportati di seguito.

Object Request Broker (ORB)

È l'elemento fondamentale dell'architettura. È il canale che fornisce l'infrastruttura che permette agli oggetti di comunicare indipendentemente dal linguaggio e dalla piattaforma adottata. La comunicazione tra tutti i componenti OMA è sempre mediata e gestita dall'ORB.

Object Services

Standardizzano la gestione e la manutenzione del ciclo di vita degli oggetti. Forniscono le interfacce base per la creazione e l'accesso agli oggetti. Sono indipendenti dal singolo dominio applicativo e possono essere usati da più applicazioni distribuite.

Common Facilities

Comunemente conosciute come CORBAFacilities. Forniscono due tipologie di servizi, orizzontali e verticali. Quelli orizzontali coprono funzionalità applicative comuni: gestione stampe, gestione documenti, database e posta elettronica. Quelli verticali sono invece destinati ad una precisa tipologia di applicazioni.

Domain Interfaces

Possono combinare common facilities e object services. Forniscono funzionalità altamente specializzate per ristretti domini applicativi.

Application Objects

È l'insieme di tutti gli altri oggetti sviluppati per una specifica applicazione. Non è un'area di standardizzazione OMG.

Di questi componenti OMG fornisce una definizione formale sia delle interfacce (mediante IDL), sia della semantica. La definizione mediante interfacce lascia ampio spazio al mercato di componenti software di agire sotto le specifiche con differenti implementazioni e consente la massima interoperabilità tra componenti diversi di case differenti.

I servizi CORBA

I CORBAServices sono una collezione di servizi system-level descritti dettagliatamente con un'interfaccia IDL; sono destinati a completare ed estendere le funzionalità fornite dall'ORB. Forniscono un supporto che va a coprire lo spettro completo delle esigenze di una qualunque applicazione distribuita. Alcuni dei servizi standardizzati da OMG (ad esempio il Naming Service) sono diventati fondamentali nella programmazione CORBA e sono presenti in tutte le implementazioni. Altri servizi appaiono invece meno interessanti nella pratica comune, assumono un significato solo dinanzi a esigenze particolari e non sono presenti nella maggior parte delle implementazioni sul mercato.

OMG ha pubblicato le specifiche di ben 15 servizi, qui riportati.

Collection Service

Fornisce meccanismi di creazione e utilizzo per le più comuni tipologie di collection.

Concurrency Control Service

Definisce un lock manager che fornisce meccanismi di gestione dei problemi di concorrenza nell'accesso a oggetti agganciati all'ORB.

Event Service

Fornisce un event channel che consente ai componenti interessati a uno specifico evento di ricevere una notifica, pur non conoscendo nulla del componente generatore.

Externalization Service

Definisce meccanismi di streaming per il trattamento dei dati da e verso i componenti.

Licensing Service

Fornisce meccanismi di controllo e verifica di utilizzo di un componente. È pensato per l'implementazione di politiche "pago per quel che uso".

Lyfe Cycle Service

Definisce le operazioni necessarie a gestire il ciclo di vita di un componente sull'ORB (creare, copiare e rimuovere).

Naming Service

Consente ad un componente di localizzare risorse (componenti o altro) mediante nome. Permette di interrogare sistemi di directory e naming già esistenti (NIS, NDS, X.500, DCE, LDAP). È il servizio più utilizzato.

Persistence Service

Fornisce mediante un'unica interfaccia le funzionalità necessarie alla memorizzazione di un componente su più tipologie di server (ODBMS, RDBMS e file system).

Properties Service

Permette la definizione di proprietà legate allo stato di un componente.

Query Service

Fornisce meccanismi di interrogazione basati su *Object Query Language*, estensione di SQL.

Relationship Service

Consente definizione e verifica dinamiche di varie associazioni e relazioni tra componenti.

Security Service

Framework per la definizione e la gestione della sicurezza in ambiente distribuito. Copre ogni possibile aspetto: autenticazione, definizione di credenziali, gestione per delega, definizione di access control list e non-repudiation.

Time Service

Definisce un'interfaccia di sincronizzazione tra componenti in ambiente distribuito.

Trader Service

Fornisce un meccanismo modello Yellow Pages per i componenti.

Transaction Service

Fornisce un meccanismo di two-phase commit sugli oggetti agganciati all'ORB che supportano il rollback; definisce transazioni flat o innestate.

Le basi CORBA

CORBA e OMA in genere si fondano su alcuni principi di design:

- separazione tra interfaccia ed implementazione: un client è legato all'interfaccia di un oggetto CORBA, non alla sua implementazione;

- Location Transparency e Access Transparency: l'utilizzo di un qualunque oggetto CORBA non presuppone alcuna conoscenza sulla sua effettiva localizzazione;

- Typed Interfaces: ogni riferimento a un oggetto CORBA ha un tipo definito dalla sua interfaccia.

In CORBA è particolarmente significativo il concetto di trasparenza, inteso sia come location transparency, sia come trasparenza del linguaggio di programmazione adottato. In pratica è trasparente al client la collocazione dell'implementazione di un oggetto, locale o remota. La location transparency è garantita dalla mediazione dell'ORB. Un riferimento ad un oggetto remoto va inteso come un identificativo unico di una sua implementazione sulla rete.

Architettura CORBA

L'architettura CORBA ruota intorno al concetto di Objects Request Broker. L'ORB è il servizio che gestisce la comunicazione in uno scenario distribuito agendo da intermediario tra gli oggetti remoti: individua l'oggetto sulla rete, comunica la richiesta all'oggetto, attende il risultato e lo comunica indietro al client. L'ORB opera in modo tale da nascondere al client tutti i dettagli sulla localizzazione degli oggetti sulla rete e sul loro linguaggio d'implementazione; è quindi l'ORB a individuare l'oggetto sulla rete e a effettuare le opportune traslazioni nel linguaggio d'implementazione. Queste traslazioni sono possibili solo per quei linguaggi per i quali è stato definito un mapping con IDL (questa definizione è stata operata per i linguaggi più comuni).

In figura 7.1 si può osservare l'architettura CORBA nel suo complesso:

- Object è l'entità composta da identity, interface e implementation (*servant* in gergo CORBA).

- Servant è l'implementazione dell'oggetto remoto. Implementa i metodi specificati dall'interfaccia in un linguaggio di programmazione.

- Client è l'entità che invoca i metodi (*operation* in gergo CORBA) del servant. L'infrastruttura dell'ORB opera in modo da rendergli trasparenti i dettagli della comunicazione remota.

- ORB è l'entità logica che fornisce i meccanismi per inviare le richieste da un client all'oggetto remoto. Grazie al suo operato, che nasconde completamente i dettagli di comunicazione, le chiamate del client sono assimilabili a semplici invocazioni locali.

- ORB Interface: essendo un'entità logica, l'ORB può essere implementato in molti modi. Le specifiche CORBA definiscono l'ORB mediante un'interfaccia astratta, nascondendo completamente alle applicazioni i dettagli d'implementazione.

- IDL stub e IDL skeleton: lo stub opera da collante tra client ed ORB; lo skeleton ha la stessa funzione per il server. Stub e skeleton sono generati nel linguaggio adottato da un compilatore apposito che opera partendo da una definizione IDL.

- Dynamic Invocation Interface (DII) è l'interfaccia che consente ad un client di inviare dinamicamente una request ad un oggetto remoto, senza conoscerne la definizione dell'interfaccia e senza avere un legame con lo stub. Consente inoltre ad un client di effettuare due tipi di chiamate asincrone: deferred synchronous (separa le operazioni di send e di receive) e oneway (solo send).

- Dynamic Skeleton Interface (DSI) è l'analogo lato server del DII. Consente ad un ORB di recapitare una request ad un oggetto che non ha uno skeleton statico, ossia non è stato definito precisamente il tipo a tempo di compilazione. Il suo utilizzo è totalmente trasparente ad un client.

- Object Adapter assiste l'ORB nel recapitare le request ad un oggetto e nelle operazioni di attivazione/disattivazione degli oggetti. Il suo compito principale è quello di legare l'implementazione di un oggetto all'ORB.

Invocazione CORBA

Utilizzando l'ORB, un client può inviare una Request in modo trasparente ad un oggetto CORBA che risieda sulla stessa macchina od ovunque sulla rete. Per raggiungere questo livello d'astrazione, ogni oggetto remoto è dotato di uno stub e di uno skeleton; questi due elementi agiscono rispettivamente da collante tra client e ORB e tra ORB ed oggetto CORBA.

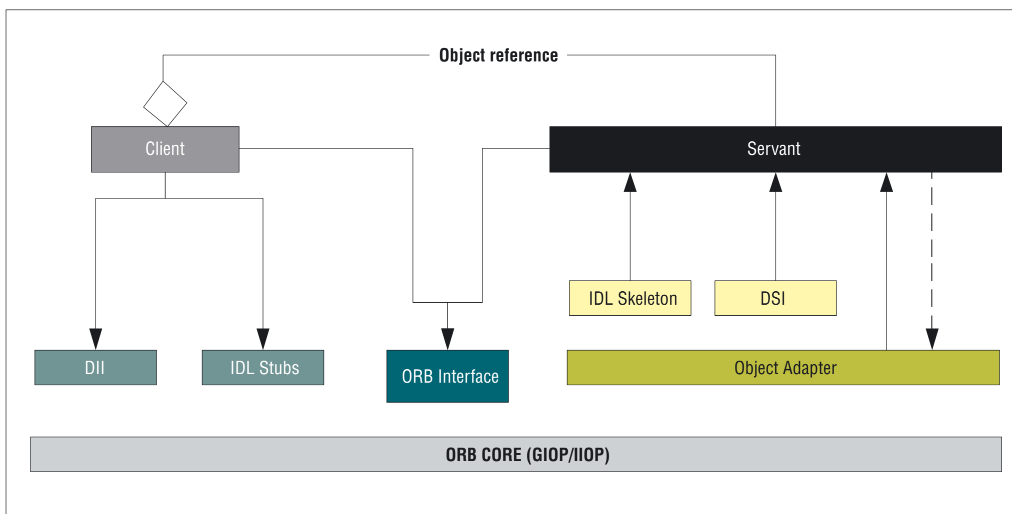
In maniera similare a quanto accade in RMI, lo stub effettua il marshalling dei dati, traslando i data types dal linguaggio di programmazione client-side a un generico formato CORBA; quest'ultimo è convogliato via rete dal messaggio di Request.

Il client invoca i metodi non sull'oggetto remoto, bensì sul suo stub locale; l'effettiva invocazione remota viene operata dallo stub. Come si vedrà più dettagliatamente in seguito, il meccanismo dello stub è una classica implementazione del pattern Proxy.

In maniera speculare a quanto effettuato dallo stub, l'unmarshalling dei dati è eseguito sul server dallo skeleton; in questo caso il formato della Request viene traslato nel linguaggio di programmazione server-side.

Come si è detto in precedenza, Stub e skeleton sono generati automaticamente da un compilatore a partire dalla definizione IDL dell'oggetto CORBA.

Figura 7.1 – *Architettura CORBA*



Interfaccia e funzionalità di un ORB

L'interfaccia di un ORB è definita dalle specifiche CORBA. La maggior parte degli ORB forniscono alcune operazioni addizionali, ma esistono alcuni metodi che dovrebbero essere forniti da tutte le implementazioni.

L'inizializzazione dell'ORB va effettuata invocando il metodo init

```
static ORB init()
static ORB init(Applet app, Properties props)
static ORB init(String[] args, Properties props)
```

Il metodo senza parametri opera secondo il pattern Singleton, restituendo un ORB di default con alcune limitazioni. Gli altri due metodi restituiscono un ORB con le proprietà specificate e sono pensati esplicitamente per le Java application e le Applet. L'array di stringhe e l'oggetto Properties consentono di impostare alcune proprietà dell'istanza di ORB restituita dal metodo init; l'array viene usato per i parametri da linea di comando. Le proprietà standard ORBClass e ORBSingletonClass consentono ad esempio di specificare l'utilizzo di un custom ORB differente da quello di default. Ogni implementazione fornisce anche proprietà aggiuntive; tutte le proprietà non riconosciute sono semplicemente ignorate. Altre funzionalità sicuramente offerte da un ORB sono le operazioni relative agli object reference. Ogni riferimento ad un oggetto (*Interoperable Object Reference*, IOR) può essere convertito in stringa; è garantito anche il processo inverso.

```
String object_to_string(Object obj)
Object string_to_object(String str)
```

Prima che un oggetto remoto sia utilizzabile da un client, va attivato sull'ORB. Come si vedrà in seguito esistono più tipologie di attivazione.

Il modo più semplice in assoluto è dato dai metodi

```
void connect(Object obj)
void disconnect(Object obj)
```

Il metodo `disconnect` disattiva l'oggetto consentendo al garbage collector di rimuoverlo.

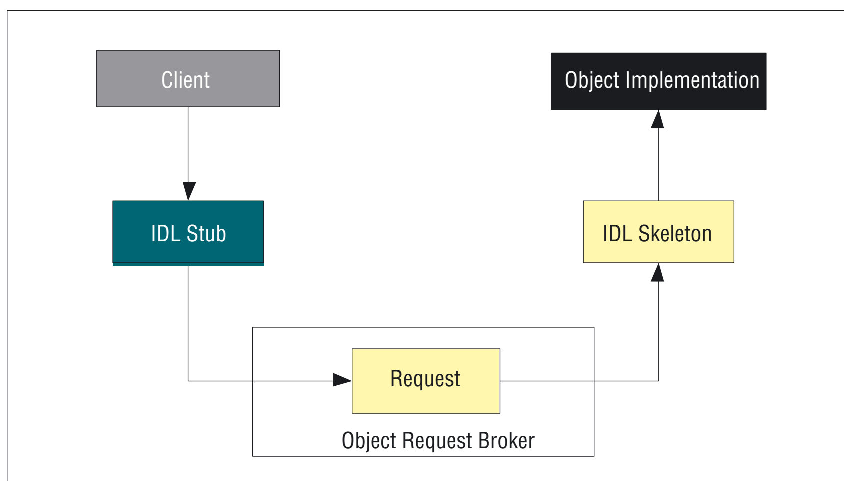
In un contesto distribuito è necessario avere a disposizione meccanismi che consentano di scoprire quali oggetti CORBA sono disponibili ed ottenere un riferimento ad essi. Anche in questo caso esistono più possibilità, la più semplice è fornita da due metodi dell'ORB

```
String[] list_initial_services()

Object resolve_initial_references(String object_name)
```

Il primo metodo elenca i servizi disponibili sull'ORB, mentre il secondo restituisce un generico riferimento ad un oggetto individuato per nome. È bene precisare che un servizio è comunque un oggetto remoto e quindi recuperabile via `resolve_initial_references`.

Figura 7.2 – *Una richiesta da client ad oggetto CORBA*



Interoperabilità tra ORB

Le specifiche CORBA 1.1 si limitavano a dare le basi per la portabilità di oggetti applicativi e non garantivano affatto l'interoperabilità tra differenti implementazioni di ORB. Le specifiche 2.0 colmarono questa significativa lacuna con la definizione di un protocollo (GIOP) espressamente pensato per interazioni ORB-to-ORB.

Il General Inter-ORB Protocol specifica un insieme di formati di messaggi e di rappresentazioni dati comuni per la comunicazione tra ORB. I tipi di dato definiti da OMG sono mappati in un messaggio di rete flat (*Common Data Representation*, CDR).

GIOP definisce i reference ad un oggetto remoto in un formato indipendente dall'ORB, l'*Interoperable Object References* (IORs). L'informazione contenuta e specificata dalla struttura dello IOR assume significato indipendentemente dall'implementazione dell'ORB, consentendo ad un'invocazione di transitare da un ORB ad un altro. Ogni ORB fornisce un metodo `object_to_string` che consente di ottenere una rappresentazione stringa dello IOR di un generico oggetto.

Vista la diffusione di TCP/IP, comunemente viene usato l'Internet Inter-ORB Protocol (IIOP) che specifica come i messaggi GIOP vengono scambiati su TCP/IP. IIOP è considerato il protocollo standard CORBA e quindi ogni ORB deve connettersi con l'universo degli altri ORB traslando le request sul e dal backbone IIOP.

Tool e implementazioni CORBA

Per realizzare un'applicazione che utilizzi il middleware definito da OMG, occorre in primo luogo disporre di un prodotto che ne fornisca un'implementazione. La garanzia teorica fornita è quella di scrivere codice utilizzabile con differenti prodotti CORBA.

Lo standard CORBA è dinamico e complesso. Di conseguenza, lo scenario dei prodotti attualmente disponibili è in continuo divenire ed il livello di aderenza dei singoli prodotti alle specifiche non è quasi mai completo. In ogni caso è sempre possibile utilizzare CORBA in maniera tale da garantire un'elevata portabilità.

Occorre comunque prestare molta attenzione alla scelta dell'ORB da utilizzare in quanto questi differiscono sia come prestazioni, sia come funzionalità fornite. Per una panoramica completa dei prodotti CORBA disponibili si veda [prodotti CORBA] in bibliografia.

Gli esempi presenti in questo capitolo fanno esplicito riferimento a due implementazioni: Sun Java IDL ed Borland VisiBroker. L'utilizzo di altri ORB con questi esempi potrebbe comportare modifiche.

Java IDL attualmente è disponibile in tre versioni decisamente differenti. L'implementazione fornita con il JDK 1.2 è limitata e il compilatore IDL va scaricato a parte da http://developer.java.sun.com/developer/earlyAccess/jdk12/idltojava.html (non tutti gli esempi presentati saranno eseguibili con questa versione). L'implementazione presente nel Java 2 SDK versione 1.3, pur essendo decisamente migliore, non implementa alcune caratteristiche importanti definite nelle specifiche (non tutti gli esempi presentati saranno eseguibili con questa versione). La versione 1.4 del Java 2 SDK fornisce un'implementazione quasi completa delle specifiche 2.3.

Borland VisiBroker è probabilmente la migliore e più diffusa implementazione CORBA presente sul mercato; è disponibile in versione trial: si veda [VisiBroker] in bibliografia.

Interface Definition Language

CORBA fornisce una chiara separazione tra l'interfaccia di un oggetto e la sua implementazione. In modo simile a quanto accade in RMI, il client non si deve occupare in modo diretto dei dettagli di implementazione, ma solo dell'interfaccia implementata dall'oggetto che intende utilizzare.

In un middleware distribuito, tutti gli oggetti, compresi quelli che lo compongono, sono trattati come interfacce. Questo è sia una valida scelta di design, sia un'esigenza di distribuzione: un client tipicamente non conosce e non deve conoscere l'implementazione di un oggetto destinato a essere eseguito su una macchina server. Questa considerazione ha una valenza ancora maggiore in un contesto tecnologico che consente ad esempio il dialogo tra oggetti Java e procedure Cobol che per natura probabilmente risiederanno addirittura su macchine ad architetture differenti.

Poiché CORBA è trasparente rispetto al linguaggio, OMG ha definito nelle sue specifiche un nuovo linguaggio interamente descrittivo (IDL) destinato alla definizione delle interfacce degli oggetti CORBA. In momenti successivi sono stati definiti i differenti mapping tra i vari linguaggi di programmazione ed IDL. È da notare che in molti dei linguaggi utilizzabili con CORBA non esiste il concetto di interfaccia (ad esempio COBOL e C).

Un oggetto remoto quindi, indipendentemente dal fatto che sia applicativo o appartenente all'infrastruttura (l'ORB, i servizi, ecc.), per essere utilizzato in un middleware CORBA deve essere in primo luogo definito mediante IDL. Nel caso di un oggetto applicativo la definizione sarà a carico dello sviluppatore, nel caso di un oggetto di infrastruttura viene fornita da OMG. Ecco ad esempio parte della definizione IDL dell'ORB:

```
// IDL
module CORBA {
  interface ORB {
    string object_to_string (in Object obj);
    Object string_to_object (in string str);
    Object resolve_initial_references (in ObjectId identifier) raises (InvalidName);

    // ecc...
  };
};
```

Sintassi e caratteristiche

La sintassi IDL è C-like e quindi è piuttosto simile anche alla sintassi Java. Sebbene sia un linguaggio descrittivo orientato agli oggetti, in modo simile al C++, IDL include la possibilità, non contemplata da Java, di definire strutture dati che non siano classi.

I blocchi logici IDL sono racchiusi in parentesi graffe; a differenza di Java è necessario terminare sempre il blocco con un " ; " e anche il singolo statement è terminato da un " ; ". Con " :: " è possibile specificare la gerarchia delle classi (equivale al " . " Java, per esempio `CORBA::Object`).

Nelle specifiche si parla di IDL come di un linguaggio case-insensitive, ma esistono implementazioni che non rispettano questa direttiva. A proposito delle regole di naming, va notato che CORBA non nasce nel mondo Java e quindi i tool IDL e le interfacce definite da OMG non rispettano le regole di naming abituali in un contesto Java.

In IDL è importante la sequenza delle definizioni dei vari elementi. Non è possibile utilizzare un elemento, sia esso una exception, una struttura dati o un'interfaccia, se non è già stato definito o almeno dichiarato; esiste in ogni caso un meccanismo di forward declaration.

IDL non implementa l'override e l'overload, queste limitazioni sono legate al fatto che molti dei linguaggi supportati non forniscono queste caratteristiche. A differenza di quanto accade in Java, in un file IDL possono esistere molte interfacce pubbliche.

IDL in pratica

La definizione IDL di un oggetto permette di specificare solo gli aspetti relativi alla sua interfaccia. Si potranno quindi definire le signature dei metodi, le eccezioni che questi rilanciano, l'appartenenza ai package, costanti e strutture dati manipolate dai metodi.

Data la definizione IDL sarà necessario utilizzare un apposito compilatore fornito a corredo dell'ORB. Dalla compilazione si otterranno un buon numero di file .java, fra cui stub, skeleton e altri contenenti codice di supporto per l'aggancio all'ORB. A partire dai file generati sarà possibile realizzare l'opportuna implementazione Java. Si provi a definire ad esempio una semplice costante in IDL

```
// IDL
module basic {
   const float PI = 3.14159;
};
```

Si compili il file IDL creato (nell'esempio basic.idl). Per la sintassi e il significato dei flag usati si rimanda alla documentazione dell'ORB.

```
idltojava -fno-cpp basic.idl
```
(per l'implementazione JDK 1.2)

```
idlj –fall basic.idl
```
(per l'implementazione J2SE 1.3 e 1.4)

```
idl2java –boa basic.idl
```
(per l'implementazione VisiBroker)

Verrà creata una sottodirectory basic e un file PI.java

```
// JAVA
package basic;
public interface PI {
   public final static float value = (float)3.14159;
}
```

La generazione del file operata dal compilatore IDL è basata sulle regole di mapping definite da OMG per il linguaggio Java.

Mapping IDL–Java

La trasposizione da linguaggio IDL a linguaggio Java effettuata dal compilatore si basa sull'insieme di regole definite da OMG che costituiscono il mapping tra i due linguaggi.

Tipi base

La definizione di regole di mapping tra IDL ed un linguaggio di programmazione implica in primo luogo la definizione di corrispondenze tra i differenti tipi di base; a runtime questo può causare errori di conversione durante il marshalling dei dati. La gestione di questi errori a runtime è a carico del programmatore.

Il problema si pone tipicamente per i tipi aventi precisione maggiore in Java che in IDL; ad esempio per i char che in Java, a differenza della maggior parte degli altri linguaggi, sono trattati come Unicode (16 bit) e non ASCII (8 bit). In IDL i tipi che trattano caratteri Unicode sono wchar e wstring. Alcuni dei tipi supportati da IDL non trovano corrispondenza in Java (ad

Tabella 7.1 – *Corrispondenza fra tipi IDL e Java>>*

Tipo IDL	Tipo Java	Eccezione
boolean	boolean	
char	char	CORBA::DATA_CONVERSION
wchar	char	CORBA::DATA_CONVERSION
octet	byte	
string	java.lang.String	CORBA::MARSHAL CORBA::DATA_CONVERSION
wstring	java.lang.String	CORBA::MARSHAL CORBA::DATA_CONVERSION
short	short	
unsigned short	short	
long	int	
unsigned long	int	
long long	long	
unsigned long long	long	
float	float	
double	double	
fixed	java.match.BigDecimal	CORBA::DATA_CONVERSION

esempio i tipi unsigned). TRUE e FALSE in IDL sono costanti e vengono mappate con i literal Java true e false.

Particolare attenzione va prestata all'utilizzo di null: CORBA non ha la nozione di null riferito alle stringhe o agli array. Un parametro stringa dovrà ad esempio essere trattato come una stringa vuota pena l'eccezione org.omg.CORBA.BadParam.

In IDL ogni tipo, base o complesso, può essere associato ad un nome mediante la parola chiave typedef; poiché in Java il concetto di alias per un tipo non esiste, nel codice generato verranno usati comunque i tipi primitivi che lo compongono.

Module e interface

Come forse si è notato nell'esempio precedente la parola chiave module viene mappata esattamente sul package Java

```
// IDL
module basic {...}

// generated Java
package basic;
```

In IDL la keyword interface permette di specificare la vera e propria interfaccia dell'oggetto remoto definendone dati membro e metodi (nel gergo CORBA attributi e operazioni). Il mapping di un'interface è ottenuto con la generazione di un'interfaccia ed alcune classi Java. Definendo la semplice interfaccia IDL

```
// IDL
module basic {

  interface HelloWorld {
    string hello();
  };

};
```

il compilatore creerà una directory basic ed una serie di file Java (usando VisiBroker o J2SE verranno generati anche altri file):

- HelloWorldImplBase è lo skeleton, la classe base per la generazione dell'oggetto remoto; fornisce i meccanismi di ricezione di una request dall'ORB e quelli di risposta;
- _HelloWorldStub è lo stub, l'implementazione client-side dell'oggetto remoto; fornisce i meccanismi di conversione tra l'invocazione del metodo e l'invocazione via ORB dell'oggetto remoto;

- HelloWorldOperations è l'interfaccia Java che contiene le signature dei metodi;
- HelloWorld è l'interfaccia Java dell'oggetto remoto, specializza HelloWorldOperations;
- HelloWorldHelper e HelloWorldHolder saranno spiegati più avanti.

L'unione delle interfacce HelloWorldOperations e HelloWorld definisce l'interfaccia dell'oggetto CORBA; sono dette rispettivamente *operations interface* e *signature interface*. Il JDK 1.2 usa vecchie regole di mapping e non genera l'interfaccia operation. La signature interface generata sarà

```
// generated Java
package basic;
public interface HelloWorld extends
HelloWorldOperations, org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity
}
```

mentre l'operations interface sarà

```
package basic;

public interface HelloWorldOperations {
  String hello ();
}
```

Come si vedrà più avanti, le altre classi serviranno come base per l'implementazione e l'utilizzo dell'oggetto remoto vero e proprio. Il linguaggio IDL supporta l'ereditarietà multipla utilizzando la normale derivazione Java tra interfacce.

```
// IDL
module basic {

  interface ClasseBaseA {
    void metodoA();
  };
  interface ClasseBaseB {
    void metodoB();
  };
  interface ClasseDerivataAB: ClasseBaseA, ClasseBaseB {
  };

};
```

ClasseDerivataAB deriva dalle altre due interfacce ed avrà quindi una rappresentazione Java.

```
// generated Java
package basic;
public interface ClasseDerivataAB extends ClasseDerivataABOperations, basic.ClasseBaseA, basic.ClasseBaseB {
}
```

un oggetto di questo tipo dovrà quindi fornire l'implementazione dei due metodi (metodoA e metodoB).

Attributi e metodi

In IDL le signature dei vari metodi sono definite con una sintassi simile a quella Java. Per comodità è possibile dare una definizione dei metodi di accesso ad un attributo (i classici get e set Java) utilizzando la keyword attribute con l'eventuale modificatore readonly.

```
// IDL
module basic {

  interface Motocicletta {
    readonly attribute string colore;
    void cambiaMarcia(in long marcia);
  };

};
```

Poiché l'attributo colore è readonly, sarà generato solo il corrispondente metodo di lettura.

```
// generated Java
package basic;
public interface MotociclettaOperations {
  String colore();
  void cambiaMarcia(int marcia);
}
```

In IDL il passaggio di parametri ad un metodo implica la dichiarazione della modalità di passaggio che si desidera adottare. A differenza di quanto accade in Java, in IDL è possibile utilizzare il passaggio per valore o quello per riferimento specificando nella signature la natura dei parametri con le keyword in, out o inout. Come si può intuire un parametro out può essere modificato dal metodo invocato.

Classi Holder

Le classi Holder sono utilizzate per supportare il passaggio di parametri out e inout. Come si è visto in precedenza, dalla compilazione di un'interfaccia IDL viene generata una corrispondente

classe <NomeInterfaccia>Holder; l'Holder è generato per ogni tipo utente. Nel package org.omg.CORBA sono forniti gli Holder per tutti i tipi primitivi. Ogni Holder fornisce un costruttore di default che inizializza il contenuto a false, 0, null o null unicode a seconda del tipo. Ecco per esempio l'Holder del tipo base int:

```
// JAVA
final public class IntHolder implements org.omg.CORBA.portable.Streamable {

  public int value;
  public IntHolder() {}
  public IntHolder(int initial) {...}

  public void _read(org.omg.CORBA.portable.InputStream is) {...}

  public void _write(org.omg.CORBA.portable.OutputStream os) {...}

  public org.omg.CORBA.TypeCode _type() {...}

}
```

Classi Helper

Per ogni tipo definito dall'utente il processo di compilazione genera una classe Helper con il nome <TipoUtente>Helper. La classe Helper è astratta e fornisce alcuni metodi statici di comodo che implementano funzionalità per manipolare il tipo associato (lettura e scrittura del tipo da/verso uno stream, lettura del repository id, e così via).

L'unica funzionalità di utilizzo comune è fornita dal metodo narrow implementato dall'Helper

```
// generated Java
package basic;
public class HelloWorldHelper {

  //...

  public static basic.HelloWorld narrow( org.omg.CORBA.Object that) throws org.omg.CORBA.BAD_PARAM {

    if (that == null)
      return null;

    if (that instanceof basic.HelloWorld)
      return (basic.HelloWorld) that;

    if (!that._is_a(id())) {
```

```
        throw new org.omg.CORBA.BAD_PARAM();
      }

      org.omg.CORBA.portable.Delegate dup = ((org.omg.CORBA.portable.ObjectImpl) that)._get_delegate();

      basic.HelloWorld result = new basic._HelloWorldStub(dup);

      return result;
    }
  }
```

Il metodo narrow effettua un cast "sicuro" dal generico Object Corba al tipo definito. Grazie ad una serie di controlli ciò che verrà ritornato sarà sicuramente un oggetto del tipo atteso oppure una Exception CORBA BAD_PARAM.

Tipi strutturati

Mediante IDL è possibile dare la definizione di entità che non siano classi o interfacce, ma semplici strutture dati. Ovviamente il mapping con Java sarà in ogni caso operato mediante classi ed interfacce.

Esistono tre categorie di tipi strutturati: enum, union e struct. Tutti i tipi strutturati sono mappati in Java con una final class fornita degli opportuni campi e costruttori, Helper e Holder.

L'enum è una lista ordinata di identificatori, la union è un incrocio tra la Union C ed un'istruzione di switch, la struct è una struttura dati che consente di raggruppare al suo interno più campi.

```
// IDL
module basic {

  enum EnumType {first, second, third, fourth, fifth};

  union UnionType switch (EnumType) {
    case first: long win;
    case second: short place;
    default: boolean other;
  };

  struct Struttura {
    string campoA;
    string campoB;
  };

};
```

Nell'esempio, Struttura sarà mappata con Helper, Holder e la classe

```
// generated Java
package basic;
public final class Struttura implements org.omg.CORBA.portable.IDLEntity {
   // instance variables
   public String campoA;
   public String campoB;
   // constructors
   public Struttura() { }
   public Struttura(String __campoA, String __campoB) {
      campoA = __campoA;
      campoB = __campoB;
   }
}
```

Sequence e array

In IDL esistono due collezioni tipizzate di dati: sequence e array. Ambedue sono mappate su array Java. Le sequence possono avere dimensioni predefinite (bounded) o non predefinite (unbounded).

```
// IDL
module basic {

   typedef sequence<octet> ByteSequence;

   typedef string MioArray[20];

   struct StrutturaConArray {
      ByteSequence campoA;
   };
};
```

La compilazione dell'esempio genererà solamente Helper e Holder per ByteSequence e MioArray. Nella struttura, il tipo ByteSequence sarà trattato come array di byte.

```
// generated Java
package basic;
public final class StrutturaConArray implements org.omg.CORBA.portable.IDLEntity {
   // instance variables
   public byte[] campoA;
   // constructors
   public StrutturaConArray() { }
```

```
    public StrutturaConArray(byte[] __campoA) {
        campoA = __campoA;
    }
}
```

Exception

La definizione di una exception in IDL non è dissimile da quella di una struct. La signature del metodo che la rilancia utilizza la keyword raises (equivalente del Java throws).

```
// IDL
module basic {

    exception MyCustomException {
        string reason;
    };
    interface HelloWorldWithException {
        string hello() raises (MyCustomException);
    };

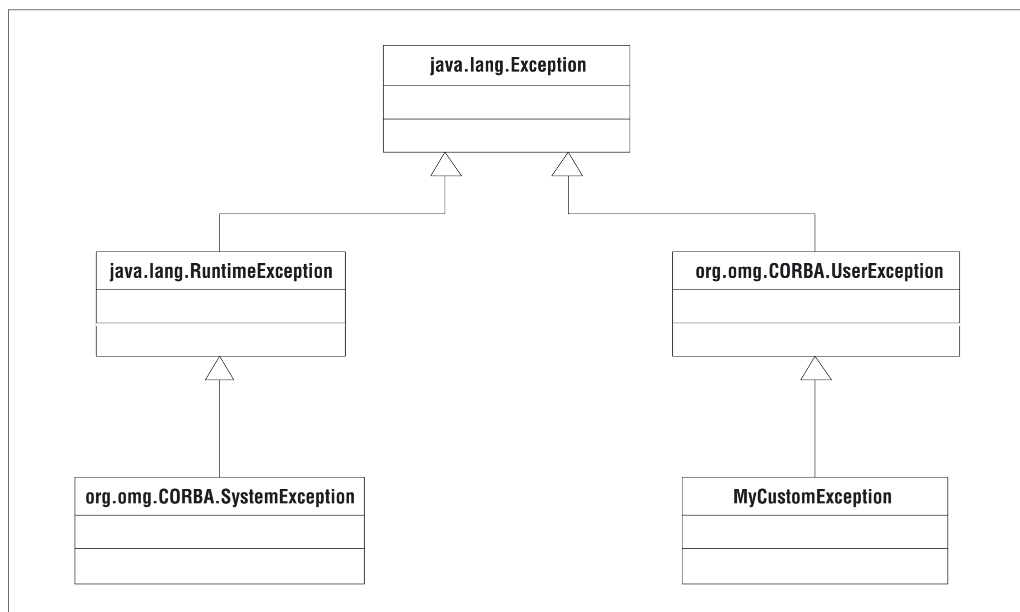
};
```

Anche il mapping Java assomiglia a quello visto nel caso di una struct, quindi una classe final con i campi definiti nell'exception ed i costruttori opportuni, più i soliti Helper e Holder

```
// generated Java
package basic;
public final class MyCustomException extends org.omg.CORBA.UserException implements
org.omg.CORBA.portable.IDLEntity {
    // instance variables
    public String reason;
    // constructors
    public MyCustomException() {
        super();
    }
    public MyCustomException(String __reason) {
        super();
        reason = __reason;
    }
}
```

Le SystemException CORBA derivano da java.lang.RuntimeException, mentre ogni UserException definita in una IDL specializza java.lang.Exception. Per questa ragione è obbligatorio l'handle or

Figura 7.3 – *Gerarchia delle eccezioni CORBA*



declare su tutte le eccezioni utente, mentre non lo è per tutte le SystemException CORBA (CORBA::MARSHAL, CORBA::OBJECT_NOT_EXIST, ecc.).

Un po' di pratica

In un caso semplice i passi da seguire per creare, esporre e utilizzare un oggetto CORBA sono i seguenti:

- descrivere mediante IDL l'interfaccia dell'oggetto che si intende implementare;
- compilare con il tool apposito il file IDL;
- identificare tra le classi e le interfacce generate quelle necessarie alla definizione dell'oggetto e specializzarle opportunamente;
- scrivere il codice necessario per inizializzare l'ORB ed informarlo circa la presenza dell'oggetto creato;
- compilare il tutto con un normale compilatore Java;

- avviare la classe di inizializzazione e l'applicazione distribuita.

Definizione IDL

Si definisca un semplice oggetto Calcolatrice che esponga un metodo in grado di computare la somma tra due numeri dati in input.

```
// IDL
module utility {
    interface Calcolatrice {
      long somma(in long a, in long b);

  };

};
```

Si compili il file Calcolatrice.idl mediante il compilatore fornito dall'ORB. Il processo di compilazione creerà una directory utility e gli opportuni file Java: _CalcolatriceImplBase, _CalcolatriceStub, CalcolatriceOperations, Calcolatrice, CalcolatriceHelper e CalcolatriceHolder. Nel caso si utilizzi il J2SE 1.4 è necessario compilare con il flag oldImplBase (es. idlj -fall -oldImplBase Calcolatrice.idl).

Implementare l'oggetto remoto

La classe base per l'implementazione è la classe astratta _CalcolatriceImplBase.java, ovvero lo skeleton.

Si può notare nella fig. 7.4 come l'implementazione dell'interfaccia "remota" Calcolatrice non sia a carico dell'oggetto remoto, bensì a carico dello skeleton.

Lo skeleton è una classe astratta e non fornisce alcuna implementazione del metodo somma definito nell'interfaccia IDL. Quindi per definire il servant CalcolatriceImpl sarà necessario specializzare _CalcolatriceImplBase e fornire l'opportuna implementazione del metodo somma.

Ecco il codice completo del servant:

```
// JAVA
package server;

import utility.*;

public class CalcolatriceImpl extends _CalcolatriceImplBase {

  public CalcolatriceImpl() {
  super();
```

```
    }

    // Implementazione del metodo remoto
    public int somma(int a, int b) {
      return a + b;
    }
  }
```

Poiché Java non supporta l'ereditarietà multipla, in alcune situazioni può essere limitante dover derivare necessariamente il servant da ImplBase. Nel caso in cui il servant debba derivare da un'altra classe è possibile utilizzare un meccanismo alternativo di delega detto Tie che non implica la specializzazione di ImplBase. In questo capitolo l'approccio Tie non sarà esaminato.

Implementare la classe Server

Si è già avuto modo di notare come nel gergo CORBA il componente remoto che espone i servizi venga definito servant. Il server invece è la classe che inizializza l'environment, istanzia l'oggetto remoto, lo rende disponibile ai client e si pone in attesa.

La classe server è quindi una classe di servizio che ha come compito fondamentale quello di creare e agganciare all'ORB l'istanza di oggetto remoto che utilizzeranno i client e di fornire a questa un contesto di esecuzione. L'inizializzazione dell'ORB è effettuata tramite il metodo init.

```
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
```

Il parametro args è semplicemente l'array di input. Saranno quindi valorizzabili da linea di comando alcune proprietà dell'ORB (per un elenco delle proprietà disponibili consultare la documentazione dell'implementazione CORBA utilizzata).

In questo primo esempio l'aggancio è effettuato senza l'ausilio di un Object Adapter. Come già anticipato, una semplice forma di "registrazione" è fornita dal metodo connect dell'ORB

```
CalcolatriceImpl calc = new CalcolatriceImpl();
orb.connect(calc);
```

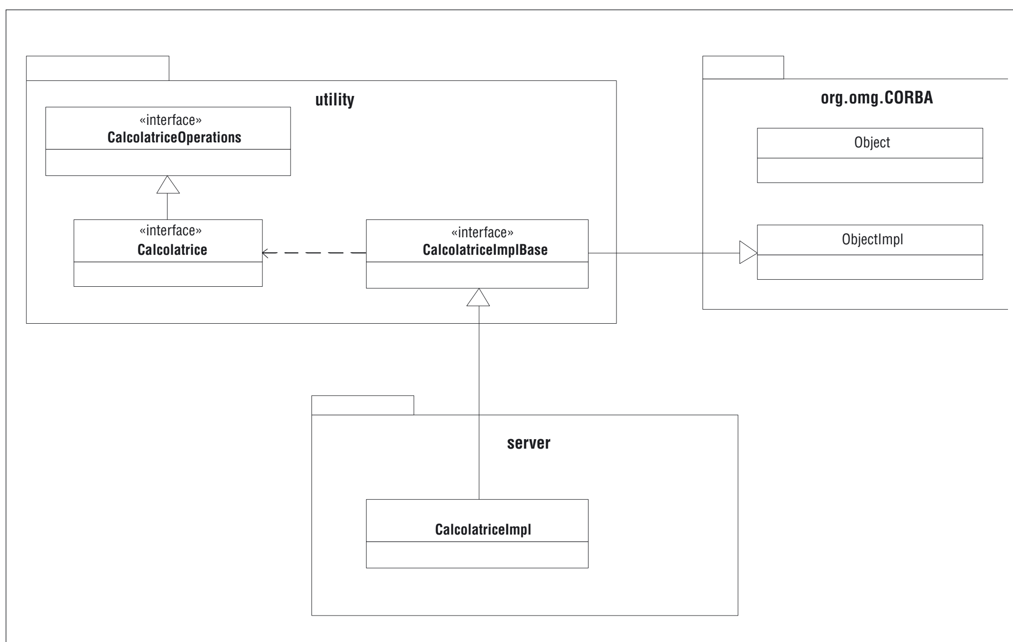
Utilizzando un meccanismo di questo tipo, il servant va considerato come un oggetto CORBA di tipo transient. Un riferimento ad un oggetto di questo tipo è valido solo nel tempo di vita di una precisa istanza del servant. Più avanti saranno analizzati gli oggetti di tipo persistent.

Per ogni ORB CORBA 2.0 compliant, l'object reference (IOR) in versione stringa è ottenibile invocando

```
orb.object_to_string(calc)
```

La stringa ottenuta è il riferimento CORBA all'istanza di calcolatrice; come tale è esattamente tutto ciò di cui necessita un client per accedere ai servizi dell'oggetto. Per fornire al client lo IOR esistono molte soluzioni, la più semplice consiste nel salvarlo su file.

Figura 7.4 – *Gerarchia di derivazione della classe servant*




```
PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter(args[0])));
out.println(orb.object_to_string(calc));
out.flush();
out.close();
```

A questo punto il server può mettersi in attesa. L'attesa è necessaria in quanto l'istanza di calcolatrice "vive" solo e soltanto nel contesto fornito dall'applicazione server; è all'interno di questa che è stato effettuato il new. Si può implementare un'attesa idle del processo server utilizzando il metodo wait di un Java Object

```
java.lang.Object sync = new java.lang.Object();
synchronized (sync) {
   sync.wait();
}
```

Ecco il codice completo della classe CalcolatriceServer.

```
// JAVA
package server;
```

```
import utility.*;
import java.io.*;

public class CalcolatriceServer {

   public static void main(String[] args) {

      if (args.length!=1) {
         System.err.println("Manca argomento: path file ior");
         return;
      }

      try {

         // Inizializza l'ORB.
         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);

         // Crea un oggetto Calcolatrice
         CalcolatriceImpl calc = new CalcolatriceImpl();
         orb.connect(calc);

         // Stampa l'object reference in versione stringa
         System.out.println("Creata Calcolatrice:\n" + orb.object_to_string(calc));
         // Scrive l'object reference nel file
         PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter(args[0])));
         out.println(orb.object_to_string(calc));
         out.close();

         // Attende l'invocazione di un client
         java.lang.Object sync = new java.lang.Object();
         synchronized (sync) {
            sync.wait();
         }

      }
      catch (Exception e) {
         System.err.println("Server error: " + e);
         e.printStackTrace(System.out);
      }
   }
}
```

Implementare il client

Il client dovrà in primo luogo inizializzare l'ORB con il metodo init(), come effettuato nella classe server.

Per ottenere il riferimento all'istanza di calcolatrice, il client dovrà leggere lo IOR memorizzato nel file generato dal server.

```
BufferedReader in = new BufferedReader(new FileReader(args[0]));
String ior = in.readLine();
in.close();
```

È possibile ottenere lo IOR invocando il metodo opposto a quello utilizzato per trasformarlo in stringa. Il metodo string_to_object fornito dall'ORB restituisce un CORBA Object a partire dalla stringa che rappresenta il suo IOR.

```
org.omg.CORBA.Object obj = orb.string_to_object(ior);
```

Il metodo string_to_object restituisce un oggetto di tipo generico e non un riferimento che consenta di invocare il metodo somma.

Per ottenere tale riferimento in uno scenario Java, si effettuerebbe un cast (in RMI, ad esempio, dopo aver effettuato una lookup si opera un cast per ottenere il tipo corretto). In un contesto CORBA invece, per convertire il generico oggetto in uno di tipo determinato, è necessario utilizzare il metodo narrow della classe <Tipo>Helper.

```
Calcolatrice calc = CalcolatriceHelper.narrow(obj);
```

A questo punto il client è in condizione di invocare il metodo remoto con le stesse modalità usate per una comune invocazione di metodo

```
calc.somma(a, b)
```

dove a e b sono da intendersi come 2 int. È da notare come questo non sia l'unico modello di invocazione CORBA. Un'invocazione di questo tipo viene detta invocazione statica, più avanti sarà affrontata l'invocazione dinamica.

Ecco il codice completo del client:

```
package client;

import utility.*;
import java.io.*;

public class CalcolatriceClient {

  public static void main(String args[]) {

    if (args.length!=1) {
      System.err.println("Manca argomento: path file ior");
      return;
```

```
}

try {

   // Crea e inizializza l'ORB
   org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);

   // Legge dal file il reference all'oggetto
   // Si assume che il server lo abbia generato
   BufferedReader in = new BufferedReader(new FileReader(args[0]));
   String ior = in.readLine();
   in.close();

   // Ottiene dal reference un oggetto remoto...
   org.omg.CORBA.Object obj = orb.string_to_object(ior);

   // ...e ne effettua il narrow a tipo Calcolatrice
   Calcolatrice calc = CalcolatriceHelper.narrow(obj);
   // Ottiene da input tastiera i 2 numeri da sommare
   BufferedReader inputUser = new BufferedReader(new InputStreamReader(System.in));
   String first, second;
   int a, b;

   // Leggo primo addendo
   System.out.println ();
   System.out.print("A = ");
   first = inputUser.readLine();
   a = Integer.valueOf(first).intValue ();

   // Leggo secondo addendo
   System.out.println ();
   System.out.print("B = ");
   second = inputUser.readLine();
   b = Integer.valueOf(second).intValue ();

   // Invoca il metodo remoto passandogli i parametri
   System.out.println ();
   System.out.print("Il risultato è: ");
   System.out.print(calc.somma(a, b));

}
catch (Exception e) {
   e.printStackTrace();
```

```
    }
  }
}
```

Eseguire l'esempio

Dopo aver compilato tutte le classi, comprese quelle generate dal precompilatore, è finalmente possibile eseguire l'esempio. La prima classe da mandare in esecuzione è la classe CalcolatriceServer

```
java server.CalcolatriceServer calc.ior
```

passando come parametro il path del file su cui si intende memorizzare lo IOR. L'output prodotto sarà

```
Creata Calcolatrice:
IOR:000000000000001d49444c3a7574696c6974792f43616c636f6c6174726963653a312e
3000000000000000010000000000000002c000100000000000004696e6b0005b70000000000018afab
cafe00000002a1ed120b000000080000000000000000
```

Si noti che, poiché il processo server si pone in attesa, l'esecuzione effettivamente non termina. Se si terminasse il processo, il client avrebbe uno IOR inservibile in quanto l'istanza identificata da questo non sarebbe più "viva". A questo punto si può mandare in esecuzione il client fornendogli il path del file generato dal server

```
java client.CalcolatriceClient calc.ior
```

Il programma richiederà in input i due numeri da sommare (si noti che non sono stati gestiti eventuali errori di conversione) e stamperà, a seguito di un'invocazione remota, il risultato.

Client e server stub

È bene effettuare una breve digressione sul concetto di stub (letteralmente "surrogato"). Lo stub compare in molti scenari di programmazione (ad esempio in DLL ed in generale nella programmazione distribuita). Esiste una precisa corrispondenza con un celebre pattern di programmazione: il pattern Proxy.

Il pattern Proxy viene tipicamente utilizzato per aggiungere un livello di indirezione. Il Proxy è un surrogato di un altro oggetto ed è destinato a "mediare" l'accesso a quest'ultimo. In generale implica la presenza di un oggetto, il Proxy, che abbia la stessa interfaccia dell'oggetto effettivo, detto Real Subject. Il Proxy riceve le richieste destinate al Real Subject e le comunica a quest'ultimo effettuando eventualmente delle operazioni prima e/o dopo l'accesso.

Osservando la fig. 7.5 è possibile vedere come lo stub e lo skeleton implementino la stessa interfaccia, quella dell'oggetto remoto. Quindi un generico client sarà in grado di dialogare con

lo stub invocando i metodi che intende far eseguire all'oggetto remoto. In tal senso lo stub opera da procuratore dell'oggetto presso il client (*proxy* significa appunto "procuratore", "delegato").

Lo stub, nella sua opera di delegato, sarà in grado di rendere invisibili al client tutti i dettagli della comunicazione remota e della locazione fisica dell'oggetto. Nell'ottica del client il dialogo sarà operato direttamente con l'oggetto remoto (questo è garantito dal fatto che lo stub implementa l'interfaccia dell'oggetto remoto). Sostituire l'implementazione non avrà alcun impatto sul client purché l'interfaccia rimanga inalterata.

Un possibile miglioramento

La soluzione proposta nell'esempio precedente è decisamente primitiva. L'accesso all'oggetto remoto è possibile solo se il client ed il server condividono una porzione di file system (in realtà sono utilizzabili anche altri meccanismi quali mail, floppy, …).

Un primo miglioramento potrebbe essere ottenuto utilizzando un Web Server. Con un Web Server attivo un client potrebbe leggere il file contenente lo IOR via HTTP. Il codice di lettura del client potrebbe essere qualcosa del genere:

```
URL urlIOR = new URL(args[0]);
DataInputStream in = new DataInputStream(urlIOR.openStream());
String ior = in.readLine();
in.close();
```

Al client andrebbe fornito non più il path, ma l'URL corrispondente al file generato dal server, ad esempio `http://localhost/corba/calc.ior`. Il server dovrà generare il file nella virtual directory corretta del Web Server (`corba` nell'esempio). Nel caso non si disponga di un Web Server è possibile scaricare gratuitamente lo "storico" Apache da `www.apache.org`.

Anche con questa modifica la soluzione, pur essendo praticabile in remoto e totalmente portabile, è ben lontana dall'ottimale. Tra i difetti che presenta è bene notare come in parte violi la location transparency promessa da CORBA: non si conosce l'effettiva collocazione dell'implementazione, ma è necessario conoscere l'URL del file IOR.

Un approccio decisamente migliore prevede l'utilizzo del CORBA Naming Service.

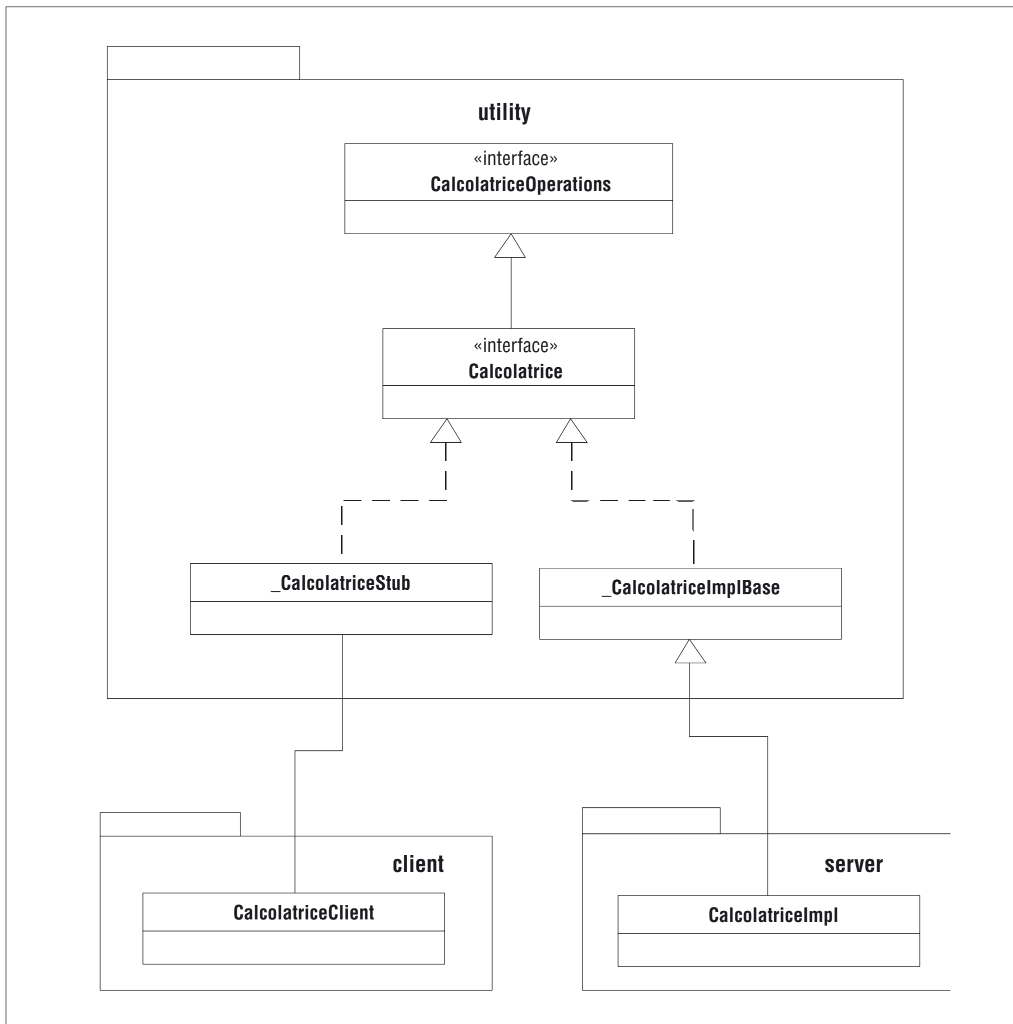
CORBA Naming Service

Il Naming Service è sicuramente il principale meccanismo CORBA per la localizzazione di oggetti su un ORB. Fa parte delle specifiche CORBA dal 1993 ed è il servizio più importante tra quelli standardizzati da OMG.

Fornisce un meccanismo di mapping tra un nome ed un object reference, quindi rappresenta anche un metodo per rendere disponibile un servant ad un client remoto. Il meccanismo è simile a quello di registrazione ed interrogazione del registry in RMI.

Nella programmazione distribuita l'utilizzo di un nome per reperire una risorsa ha degli evidenti vantaggi rispetto all'utilizzo di un riferimento. In primo luogo il nome è significativo

Figura 7.5 – *Il pattern Proxy e gli stub server e client*



per lo sviluppatore, in secondo luogo è completamente indipendente dagli eventuali restart dell'oggetto remoto.

Struttura del Naming Service

L'idea base del Naming Service è quella di incapsulare in modo trasparente i servizi di naming e directory già esistenti. I nomi quindi sono strutturabili secondo uno schema gerar-

chico ad albero, uno schema astratto indipendente dalle singole convenzioni delle varie piattaforme di naming o di directory. L'operazione che associa un nome ad un reference è detta bind (esiste anche l'operazione inversa di unbind). L'operazione che recupera un reference a partire da un nome è detta naming resolution.

Esistono più naming context; all'interno di questi il nome è univoco. I naming context possono essere ricondotti ai nodi intermedi dell'albero di naming ed al concetto di directory in un file system. Possono esistere più nomi associati ad uno stesso oggetto.

In questo scenario quindi un nome è una sequenza di name components; questi formano il cosiddetto compound name. I nodi intermedi sono utilizzati per individuare un context, mentre i nodi foglia sono i simple name.

Un compound name quindi individua il cammino (path) che, attraverso la risoluzione di tutti i context, porta al simple name che identifica la risorsa.

Ogni `NameComponent` è una struttura con due elementi. L'identifier è la stringa nome, mentre il kind è un attributo associabile al component. Questo attributo non è considerato dal Naming Service, ma è destinato al software applicativo.

La definizione IDL del `NameComponent` è la seguente

```
// IDL
module CosNaming {

  struct NameComponent {
    Istring id;
    Istring kind;
  };

  typedef sequence <NameComponent> Name;
}
```

L'interfaccia principale è `NamingContext` e fornisce tutte le operazioni necessarie alla definizione dell'albero di Naming ed alla sua navigazione. Per quel che concerne il bind vengono fornite due funzionalità di bind Name – Object, due funzionalità di bind Name – Naming Context ed una funzionalità di unbind.

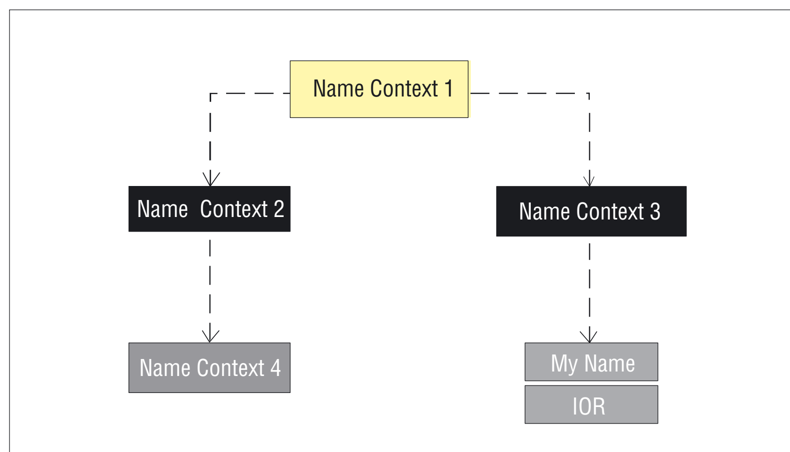
```
// IDL
module CosNaming{

  // ...

  interface NamingContext
  {

    // ...
    void bind(in Name n, in Object obj) raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
```

Figura 7.6 – *Struttura ad albero di naming*




```
      void rebind(in Name n, in Object obj) raises(NotFound, CannotProceed, InvalidName);

      void bind_context(in Name n, in NamingContext nc) raises(NotFound, CannotProceed, InvalidName, AlreadyBound);

      void rebind_context(in Name n, in NamingContext nc) raises(NotFound, CannotProceed, InvalidName);

      void unbind(in Name n) raises(NotFound, CannotProceed, InvalidName);

    };
  };
```

I metodi rebind differiscono dai metodi bind semplicemente nel caso in cui il nome sia già presente nel context; rebind sostituisce l'object reference mentre bind rilancia un'eccezione AlreadyBound.

La definizione dell'albero implica anche la possibilità di creare o distruggere i NamingContext.

```
// IDL
module CosNaming{

  // ...

  interface NamingContext
  {

    // ...
```

```
        NamingContext new_context();

        NamingContext bind_new_context(in Name n) raises(NotFound, AlreadyBound, CannotProceed, InvalidName);

        void destroy() raises(NotEmpty);

    };
};
```

La risoluzione di un name è attuata mediante il metodo resolve. La procedura di risoluzione di un nome gerarchico implicherà la navigazione ricorsiva dell'albero di context.

```
// IDL
module CosNaming{
    // ...

    interface NamingContext
    {

        // ...

        Object resolve(in Name n) raises(NotFound, CannotProceed, InvalidName);

    };
};
```

La navigazione del Naming è invece legata all'utilizzo del metodo list che ritorna un insieme di name sui quali è possibile operare iterativamente. Più precisamente ciò che viene ritornato è un oggetto di tipo BindingIterator che, tramite i metodi next_one e next_n, permette di navigare attraverso tutti i bind associati al context.

```
// IDL
module CosNaming{

    //...

    interface BindingIterator {

        boolean next_one(out Binding b);

        boolean next_n(in unsigned long how_many, out BindingList bl);

        void destroy();
```

```
    };

    interface NamingContext
    {

      // ...

      void list(in unsigned long how_many, out BindingList bl, out BindingIterator bi);

    };
  };
```

Utilizzare il Naming Service

Alla luce di quanto visto finora sul Naming Service, è possibile migliorare l'esempio della calcolatrice visto in precedenza. Lo scenario generale non cambia: esiste un oggetto servant (non necessita di alcuna modifica), un oggetto server di servizio (pubblica il servant) ed il client.

Il Naming Service ha impatto solo sulle modalità di registrazione e di accesso all'oggetto, quindi solo sul client e sul server. Anche l'IDL non necessita di alcuna modifica.

Per pubblicare un oggetto, il server dovrà in primo luogo ottenere un riferimento al Naming Service. Come detto in precedenza, è possibile ottenere un riferimento ad un qualunque servizio CORBA invocando sull'ORB il metodo `resolve_initial_references`. Ottenuto il riferimento, come per qualunque oggetto CORBA, andrà utilizzato il `narrow` fornito dal corrispondente Helper.

```
org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");

NamingContext ncRef = NamingContextHelper.narrow(objRef);
```

A questo punto è possibile effettuare il bind tra il `NameComponent` opportuno e l'istanza di servant (`calc` è in questo caso l'istanza di `CalcolatriceImpl`; si veda più avanti il codice completo).

```
NameComponent nc = new NameComponent("Calc", " ");
NameComponent path[] = {nc};
ncRef.rebind(path, calc);
```

I due parametri del costruttore `NameComponent` sono, rispettivamente, il name ed il kind.

Ecco il codice completo della classe server

```
package server;

import utility.*;
```

```
import java.io.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;

public class CalcolatriceServer {

  public static void main(String[] args) {

    try {

      // Inizializza l'ORB.
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);

      // Crea un oggetto Calcolatrice
      CalcolatriceImpl calc = new CalcolatriceImpl();
      orb.connect(calc);

      // Stampa l'object reference in versione stringa
      System.out.println("Creata Calcolatrice:\n" + orb.object_to_string(calc));

      // Root naming context
      org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");

      NamingContext ncRef = NamingContextHelper.narrow(objRef);

      // Fa il bind dell'oggetto nel Naming
      NameComponent nc = new NameComponent("Calc", " ");
      NameComponent path[] = {nc};
      ncRef.rebind(path, calc);

      // Attende l'invocazione di un client
      java.lang.Object sync = new java.lang.Object();
      synchronized (sync) {
        sync.wait();
      }

    } catch (Exception e) {
      System.err.println("Server error: " + e);
      e.printStackTrace(System.out);
    }
  }
}
```

Il client dovrà effettuare le stesse operazioni per ottenere il riferimento al Naming Service. Poi dovrà effettuare la resolve per ottenere un riferimento all'oggetto remoto Calcolatrice.

```
NameComponent nc = new NameComponent("Calc", " ");
NameComponent path[] = {nc};
Calcolatrice calc = CalcolatriceHelper.narrow(ncRef.resolve(path));
```

Ecco il codice completo della classe client.

```
package client;

import utility.*;
import java.io.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class CalcolatriceClient {

  public static void main(String args[]) {

    try {

      // Crea e inizializza l'ORB
      ORB orb = ORB.init(args, null);

      // Root naming context
      org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");

      NamingContext ncRef = NamingContextHelper.narrow(objRef);

      // Utilizzo il Naming per ottenere il riferimento all'oggetto
      NameComponent nc = new NameComponent("Calc", "");
      NameComponent path[] = {nc};
      Calcolatrice calc = CalcolatriceHelper.narrow(ncRef.resolve(path));

      // Ottiene da input tastiera i 2 numeri da sommare
      BufferedReader inputUser = new BufferedReader (new InputStreamReader(System.in));
      String first, second;
      int a, b;

      // Leggo primo addendo
      System.out.println ();
```

```
            System.out.print("A = ");
            first = inputUser.readLine();
            a = Integer.valueOf(first).intValue ();

            // Leggo secondo addendo
            System.out.println ();
            System.out.print("B = ");
            second = inputUser.readLine();
            b = Integer.valueOf(second).intValue ();

            // Invoca il metodo remoto passandogli i parametri
            System.out.println ();
            System.out.print("Il risultato è: ");
            System.out.print(calc.somma(a, b));

        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Per eseguire l'esempio sarà necessario effettuare un passo in più rispetto a quanto fatto in precedenza, ovvero attivare il Naming Service prima di mandare in esecuzione il server.

L'attivazione del servizio è legata all'implementazione CORBA, sarà quindi differente da ORB a ORB (se si utilizza VisiBroker è necessario avviare anche il tool OsAgent trattato in seguito).

`tnameserv -ORBInitialPort 1050` (per l'implementazione Sun)

`nameserv NameService` (per l'implementazione VisiBroker)

A questo punto sarà possibile avviare il server (per il significato dei flag utilizzati si rimanda alla documentazione del prodotto).

`java server.CalcolatriceServer -ORBInitialPort 1050` (per l'implementazione Sun)

`java –DSVCnameroot =NameService server.CalcolatriceServer` (per l'implementazione VisiBroker)

E infine avviare il client

`java client.CalcolatriceClient -ORBInitialPort 1050` (per l'implementazione Sun)

`java –DSVCnameroot =NameService client.CalcolatriceClient` (per l'implementazione VisiBroker)

È possibile notare come in questa modalità l'utilizzo di CORBA abbia parecchie similitudini con quello di RMI: l'utilizzo di tnameserv (nameserv) rimanda a quello di rmiregistry, così come i metodi del NamingContext bind e rebind rimandano ai metodi usati per RMI Naming.bind e Naming.rebind. Una volta ottenuto il reference, l'utilizzo dell'oggetto remoto è sostanzialmente identico.

Accesso concorrente a oggetti remoti

Uno dei compiti più complessi nell'ambito della programmazione distribuita è l'implementazione e la gestione della concorrenza. Un oggetto remoto distribuito sulla rete può essere utilizzato contemporaneamente da più client. Questo è uno dei fattori che rendono allettante la programmazione distribuita poiché l'utilizzo in concorrenza di una qualunque risorsa implica un suo migliore sfruttamento.

La gestione della concorrenza in ambiente distribuito è strettamente legata al design dell'applicazione e tipicamente va a ricadere in uno dei tre approcci qui riportati.

- unico thread: il thread è unico, le richieste sono gestite in modo sequenziale ed eventualmente accodate;
- un thread per client: viene associato un thread alla connessione con il client; le successive richieste del client sono a carico di questo thread;
- un thread per request: esiste un pool di thread utilizzati in modo concorrente per rispondere alle richieste dei client.

Esisteranno comunque situazioni in cui i differenti thread dovranno accedere a una risorsa condivisa (p.e.: connessione a DB, file di log, …); in questi casi l'accesso alla risorsa andrà opportunamente sincronizzato.

Come si vedrà più avanti, con CORBA è possibile specificare le politiche di gestione dei thread mediante l'utilizzo degli object adapter. Nella pratica, questi aspetti vengono spesso gestiti applicativamente con l'adozione di opportuni pattern di programmazione.

Questo permette di realizzare soluzioni complesse del tutto indipendenti dall'implementazione e dalla tecnologia. Tipicamente queste soluzioni sono attuate con l'adozione del pattern Factory.

Il pattern Factory

Quando si istanzia un oggetto è necessario fare un riferimento diretto ed esplicito a una precisa classe, fornendo gli eventuali parametri richiesti dal costruttore. Ciò vincola implicitamente l'oggetto utilizzatore all'oggetto utilizzato. Se questo può essere accettabile nella maggior parte dei casi, talvolta è un limite troppo forte.

In questi casi può essere molto utile incapsulare in una classe specializzata tutte le valutazioni relative alla creazione dell'oggetto che si intende utilizzare. Questa soluzione è il più noto tra

i creational patterns: il pattern Factory (anche detto Factory Method o Virtual Constructor). *Factory* letteralmente vuol dire "fabbrica" ed è proprio questo il senso dell'oggetto Factory: fabbricare sulla base di alcune valutazioni un determinato oggetto. Più formalmente, facendo riferimento anche alla fig. 7.7, un oggetto Factory fabbrica oggetti ConcreteProduct appartenenti a una determinata famiglia specificata dalla sua interfaccia (o classe astratta) Product.

Un client non crea mai in modo diretto un'istanza del Product (in un contesto remoto probabilmente non ne conoscerà nemmeno la classe), ma ne ottiene un'istanza valida attraverso l'invocazione del FactoryMethod sull'oggetto Factory. In questo modo è possibile sostituire in ogni momento l'implementazione ConcreteProductA con un'implementazione omologa ConcreteProductB senza che un eventuale client se ne accorga.

In realtà i vantaggi elencati in precedenza sono già impliciti in uno scenario CORBA (il disaccoppiamento è garantito dalla funzionalità Proxy dello stub). Nella programmazione distribuita il pattern Factory ha però altri vantaggi, in particolare consente di implementare meccanismi di load-balancing e fault-tolerance.

Poiché è la Factory a determinare la creazione del Product, essa potrà:

- istanziare un oggetto per ogni client;
- applicare un round-robin tra le differenti istanze già create, ottenendo un semplice load-balancing;
- restituire differenti tipologie di oggetti appartenenti alla famiglia Product sulla base di valutazioni legate all'identità del client;
- implementare un semplice fault-tolerance, escludendo dal pool di oggetti quelli non più funzionanti o non più raggiungibili via rete.

Un esempio di Factory

Per sperimentare un tipico caso di applicazione del pattern Factory si realizzi il classico "carrello della spesa". Per l'esempio saranno implementate la classe carrello (ShoppingCart) e una sua Factory (ShoppingCartFactory). La prima fornirà una funzionalità di acquisto e una di restituzione del contenuto, la seconda sarà dotata del solo metodo getShoppingCart.

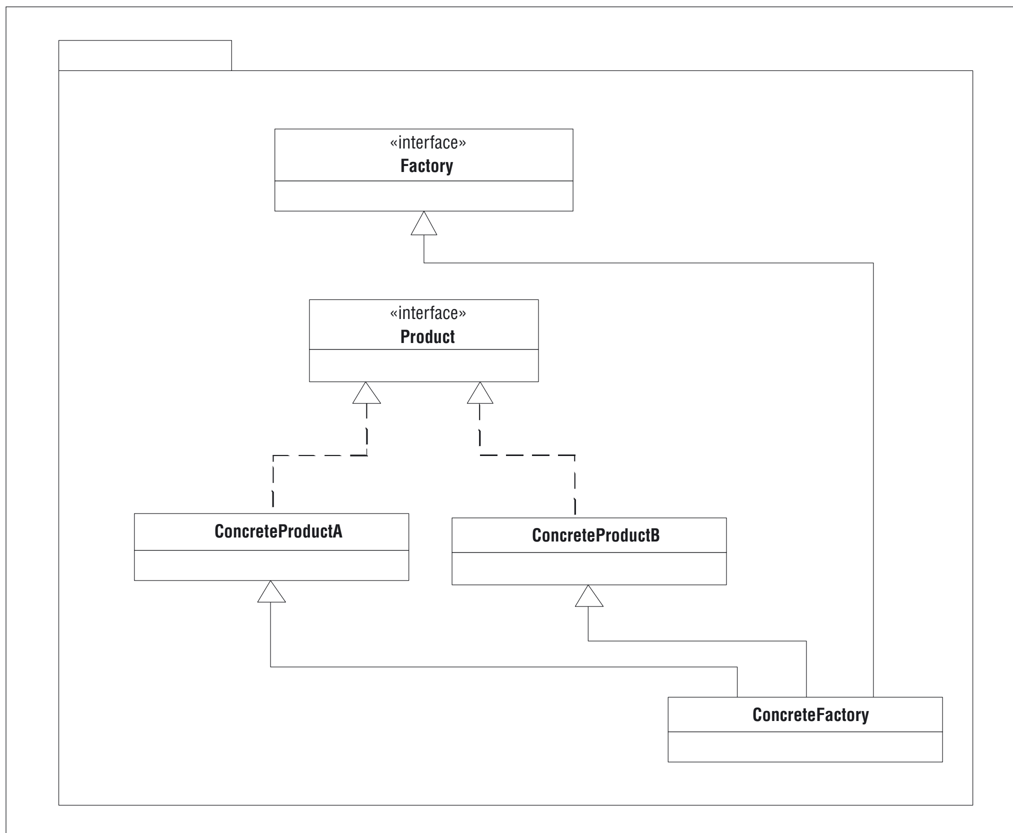
Si definisce l'IDL

```
// IDL
module shopping {

  struct Book {
    string Author;
    string Title;
  };

  typedef sequence <Book> BookList;
```

Figura 7.7 – *Il pattern Factory*




```
    interface ShoppingCart {
        void addBook(in Book book);
        BookList getBookList();
    };

    interface ShoppingCartFactory {
        ShoppingCart getShoppingCart(in string userID);
    };
};
```

Sono definite come interface sia la Factory (con il Factory Method getShoppingCart), sia il carrello vero e proprio. La Factory può essere considerata una classe di infrastruttura, mentre tutti i metodi di business sono implementati nello ShoppingCart. La classe ShoppingCartImpl imple-

menta i due semplici metodi di business e non presenta nulla di nuovo rispetto a quanto visto in precedenza.

```
package server;

import shopping.*;
import java.util.Vector;

public class ShoppingCartImpl extends _ShoppingCartImplBase {

  Vector v = new Vector();
  public ShoppingCartImpl() {
    super();
  }

  // Aggiunge un libro al carrello
  public void addBook(Book book) {
    v.add(book);
  }

  // Restituisce l'elenco dei libri acquistati
  public Book[] getBookList() {
    Book[] books = new Book[v.size()];

    for (int i=0; i<v.size(); i++)
      books[i] = (Book) v.elementAt(i);

    return books;
  }
}
```

Più interessante è l'oggetto Factory che ha il compito di generare le istanze di ShoppingCartImpl da assegnare ai client. Nel metodo getShoppingCart viene stabilita la politica di creazione e restituzione delle istanze di carrello, nel caso in esame la decisione è ovvia in quanto il carrello ha evidentemente un rapporto uno a uno con i client.

Per memorizzare le varie istanze si usa un oggetto di tipo Dictionary. Alla prima connessione dell'utente Factory creerà il carrello e lo "registrerà" sull'ORB. La Factory può ottenere un riferimento valido all'ORB invocando il metodo _orb() fornito da org.omg.CORBA.Object. Ecco la classe Factory.

```
package server;

import shopping.*;
import java.util.*;
```

```
public class ShoppingCartFactoryImpl extends _ShoppingCartFactoryImplBase {

  private Dictionary allCarts = new Hashtable();

  public ShoppingCartFactoryImpl() {

    super();
  }
  public synchronized ShoppingCart getShoppingCart(String userID) {

    // Cerca il carrello assegnato allo userID...
    shopping.ShoppingCart cart
    = (shopping.ShoppingCart) allCarts.get(userID);

    // ...se non lo trova...
    if(cart == null) {

      // Crea un nuovo carrello...
      cart = new ShoppingCartImpl();

      // ...e lo attiva sull'ORB
      _orb().connect(cart);

      System.out.println("Created " + userID + "'s cart: " + cart);

      // Salva nel dictionary associandolo allo userID
      allCarts.put(userID, cart);
    }

    // Restituisce il carrello
    return cart;
  }
}
```

È da notare che la Factory sarà utilizzata in concorrenza da più client, di conseguenza sarà opportuno sincronizzare il metodo getShoppingCart per ottenere un'esecuzione consistente.

Per quanto detto in precedenza, un client otterrà un oggetto remoto di tipo ShoppingCart interagendo con la Factory. Pertanto, l'unico oggetto registrato sul Naming Service sarà l'oggetto Factory. La registrazione sarà effettuata dalla classe server con il nome ShoppingCartFactory con le stesse modalità viste nell'esempio precedente (il codice non viene qui mostrato).

Dopo aver ottenuto il reference allo ShoppingCart assegnato, il client potrà operare direttamente su questo senza interagire ulteriormente con la Factory.

```
package client;

import shopping.*;
```

```
import java.io.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class ShoppingCartClient {
  public static void main(String args[]) {

    if (args.length != 3) {
      System.err.println("Uso corretto: java ShoppingCartClient userId Autore Titolo");
      return;
    }

    try{

      // Crea e inizializza l'ORB
      ORB orb = ORB.init(args, null);

      // Root naming context
      org.omg.CORBA.Object objRef
      = orb.resolve_initial_references("NameService");

      NamingContext ncRef = NamingContextHelper.narrow(objRef);

      // Utilizzo il Naming per ottenere il
      // riferimento all'oggetto Factory
      NameComponent nc = new NameComponent("ShoppingCartFactory", " ");
      NameComponent path[] = {nc};
      ShoppingCartFactory factory
      = ShoppingCartFactoryHelper.narrow(ncRef.resolve(path));

      // Ottengo dalla Factory un'oggetto ShoppingCart
      ShoppingCart cart = factory.getShoppingCart(args[0]);

      // Aggiungo un libro
      cart.addBook(new Book(args[1],args[2]));

      // Ottengo la lista dei libri e la stampo
      Book[] list = cart.getBookList();

      for(int i=0; i<list.length; i++)
        System.out.println("Autore " + list[i].Author + " - Titolo " + list[i].Title);

    } catch (Exception e) {
```

```
            e.printStackTrace();
        }
    }
}
```

I passi necessari per l'esecuzione sono gli stessi visti nell'esempio precedente. Al client vanno "passati" lo userId, l'autore e il titolo del libro da aggiungere al carrello.

L'adozione di un pattern è una scelta del tutto indipendente dalla tecnologia e quindi adattabile a qualunque altro scenario (ad esempio RMI). Un design come quello visto rende l'architettura più robusta e adattabile, consentendo di modificare e configurare il comportamento anche a start up avvenuto senza alcun impatto sui client. In ottica Enterprise la resistenza ai cambiamenti è di massimo interesse. Come già accennato, le politiche relative alla gestione delle istanze sono configurabili anche utilizzando gli Object Adapter.

Utilizzo degli Object Adapter

L'Object Adapter è un componente molto importante dell'architettura CORBA. Uno dei suoi compiti è quello di associare un riferimento a una specifica implementazione nel momento in cui un oggetto è invocato. Quando un client effettua un'invocazione l'adapter collabora con l'ORB e con l'implementazione per fornire il servizio richiesto.

Se un client chiama un oggetto che non è effettivamente in memoria, l'adapter si occupa anche di attivare l'oggetto affinché questo possa rispondere all'invocazione. In molte implementazioni l'adapter può occuparsi anche di disattivare un oggetto non utilizzato da lungo tempo. Dal punto di vista del client l'implementazione è sempre disponibile e caricata in memoria. Formalmente le specifiche CORBA individuano per l'adapter sei funzionalità chiave:

- Generazione e interpretazione degli object references.
- Invocazione dei metodi attraverso lo skeleton.
- Sicurezza delle interazioni.
- Autenticazione alla chiamata (utilizzando un'entità CORBA detta Principal).
- Attivazione e disattivazione degli oggetti.
- Mapping tra reference e corrispondente implementazione.
- Registrazione dell'implementazione.

Queste funzionalità sono associate al componente logico adapter e nella pratica sono compiute in collaborazione con il core dell'ORB ed eventualmente con altri componenti; alcune

funzionalità sono delegate integralmente all'ORB e allo skeleton. L'adapter è comunque coinvolto in ogni invocazione di metodo.

Le funzionalità dell'adapter rendono disponibile via ORB l'implementazione del CORBA object e supportano l'ORB nella gestione del runtime environment dell'oggetto. Dal punto di vista del client l'adapter è il componente che garantisce che le sue richieste siano recapitate a un oggetto attivo in grado di soddisfare la richiesta.

Il meccanismo CORBA opera in modo tale da consentire l'utilizzo contemporaneo di più tipi di adapter con differenti comportamenti (nelle specifiche gli adapter vengono definiti pluggable). A livello di design, l'idea di individuare un'altra entità come l'adapter nasce dalla necessità di modellare in maniera flessibile alcuni aspetti senza estendere l'interfaccia dell'ORB, ma individuando nuovi moduli pluggable.

Il primo tipo di adapter introdotto da OMG è il *Basic Object Adapter* o BOA. Poiché le specifiche del BOA erano lacunose (non definivano ad esempio i meccanismi di attivazione e disattivazione degli oggetti), i vari venditori finirono per realizzarne implementazioni proprietarie largamente incompatibili tra loro che minavano di fatto la portabilità lato server di CORBA. Per questa ragione OMG decise di abbandonare il BOA specificando un nuovo tipo di adapter, il *Portable Object Adapter* o POA.

Nel caso in cui l'ORB scelto supporti il POA sarà sicuramente opportuno utilizzarlo. Esistono tuttavia molti ORB che attualmente non forniscono un'implementazione del POA. Per tale ragione sarà esaminato anche il BOA nonostante il suo utilizzo sia formalmente deprecato. Per gli esempi di questa sezione non sarà possibile utilizzare l'implementazione Sun che non fornisce BOA e POA.

Basic Object Adapter (BOA)

Le specifiche CORBA elencano quali compiti primari del BOA la creazione/distruzione degli object reference e il reperimento delle informazioni a questi correlate. Nelle varie implementazioni il BOA, per compiere le sue attività, può accedere al componente proprietario Implementation Repository.

Come si è detto in precedenza, BOA va pensato come un'entità logica e in effetti alcuni dei suoi compiti sono svolti in cooperazione con altri componenti (ad esempio la creazione e la distruzione di object reference sono a carico dello skeleton). Questo ha un impatto sulla sua implementazione che solitamente suddivide i suoi compiti tra il processo ORB, il codice generato dal compilatore IDL e l'effettivo BOA. Comunque il BOA fornisce l'interfaccia con i metodi necessari a registrare/deregistrare gli oggetti e ad avvertire l'ORB che l'oggetto è effettivamente pronto a rispondere alle invocazioni.

Si è già visto negli esempi precedenti il concetto di server: il server è un'entità eseguibile separata che attiva l'oggetto e gli fornisce un contesto di esecuzione. Anche il BOA per attivare un oggetto si appoggia a un server.

Il server può essere attivato on demand dal BOA (utilizzando informazioni contenute nell'Implementation Repository) oppure da qualche altra entità (ad esempio uno shell script). In ogni caso il server attiverà l'implementazione chiamando il metodo `obj_is_ready` oppure il metodo `impl_is_ready` definiti nel seguente modo

```
// PIDL
module CORBA {
  interface BOA {

    void impl_is_ready (in ImplementationDef impl);

    void deactivate_impl (in ImplementationDef impl);

    void obj_is_ready (
      in Object obj, in ImplementationDef impl
    );

    void deactivate_obj (in Object obj);

    //...altri metodi di generazione references e access control
  };
};
```

In quasi tutti gli ORB, `obj_is_ready` è il metodo di registrazione del singolo oggetto all'interno di un server e stabilisce un'associazione tra un'istanza e un'entità nell'Implementation Repository.

Il metodo `impl_is_ready` è comunemente implementato come un loop infinito che attende le request del client; il ciclo non cessa fino a quando non viene invocato il `deactivate_impl`.

Esistono molti modi di combinare un server process con l'attivazione di oggetti (un server registra un oggetto, un server registra *n* oggetti, …). Le specifiche CORBA individuano quattro differenti politiche di attivazione.

- Shared Server: il processo server inizializza più oggetti invocando per ognuno `obj_is_ready`. Al termine di queste inizializzazioni il server notifica al BOA, con `impl_is_ready`, la sua disponibilità e rimane attivo fino all'invocazione di `deactivate_impl`. Gli oggetti possono essere singolarmente disattivati con `deactivate_obj`. La disattivazione è quasi sempre automatizzata dal distruttore dello skeleton.

- Unshared Server: ogni oggetto viene associato a un processo server differente. L'inizializzazione avviene comunque con le due chiamate `obj_is_ready` e `impl_is_ready`.

- Server-per-method: un nuovo processo viene creato ad ogni invocazione. Il processo termina al terminare dell'invocazione e, poiché ogni invocazione implica un nuovo processo, non è necessario inviare una notifica al BOA (alcuni ORB richiedono comunque l'invocazione di `impl_is_ready`).

- Persistent Server: tipicamente è un processo avviato mediante qualche meccanismo esterno al BOA (shell script o avvio utente) e va registrato mediante `impl_is_ready`. Dopo la notifica al BOA si comporta esattamente come uno shared server.

A differenza di quanto indicato nelle specifiche, la maggior parte delle implementazioni fornisce un sottoinsieme delle activation policy. In generale l'utilizzo dei metodi legati al BOA è differente tra i vari ORB e genera quasi sempre problemi di portabilità per quanto concerne l'attivazione delle implementazioni.

BOA in pratica

Si provi ora a riscrivere l'applicazione ShoppingCart utilizzando l'implementazione BOA fornita da VisiBroker. Per quanto detto sugli adapter si dovrà intervenire sulle classi coinvolte nell'attivazione e nell'invocazione degli oggetti CORBA, andranno quindi modificate le seguenti classi: ShoppingCartServer, ShoppingCartFactoryImpl e ShoppingCartClient.

Si utilizzi il modello di server persistent, una classe Java con metodo main lanciata da linea di comando o da script. La registrazione dell'oggetto Factory sarà effettuata via BOA.

Il BOA è un cosiddetto pseudo-object ed è possibile ottenere un riferimento valido ad esso mediante l'invocazione di un metodo dell'ORB: BOA_init. Nell'implementazione VisiBroker esistono due differenti metodi BOA_init che permettono di ricevere un BOA inizializzato con differenti politiche di gestione dei thread (thread pooling o per session) e di trattamento delle comunicazioni (utilizzo di Secure Socket Layer o no).

Invocando il metodo BOA_init senza parametri si otterrà un BOA con le politiche di default (thread pooling senza SSL). Il codice completo della classe server è

```
package server;

import shopping.*;

public class ShoppingCartServer {

  public static void main(String[] args) {

  // Inizializza l'ORB
  org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);

  // Crea l'oggetto Factory
  ShoppingCartFactoryImpl factory
  = new ShoppingCartFactoryImpl("ShoppingCartFactory");

  // Inizializza il BOA
  // N.B. Utilizzo classi proprietarie
  com.inprise.vbroker.CORBA.BOA boa = ((com.inprise.vbroker.CORBA.ORB)orb).BOA_init();

  // Esporta l'oggetto factory
  boa.obj_is_ready(factory);
```

```
    System.out.println(factory + " is ready.");

    // Attende le requests
    boa.impl_is_ready();

  }
}
```

Si noti che istanziando l'oggetto Factory si è fornita al costruttore la stringa "ShoppingCartFactory". In VisiBroker, specificando un object name quando si istanzia l'oggetto, si ottiene un riferimento di tipo persistent. Il costruttore dell'oggetto dovrà comunque notificare il nome alla superclasse.

```
public ShoppingCartFactoryImpl(String name) {
  super(name);
}
```

L'associazione tra un reference persistent e il nome viene registrata in un tool proprietario denominato OSAgent o ORB Smart Agent.

Pur essendo uno strumento proprietario OSAgent è molto utilizzato nella pratica. Fornisce una versione semplificata di naming service (con VisiBroker viene fornita anche un'implementazione standard del servizio) e implementa alcuni meccanismi proprietari di fault-tolerance e load-balancing. Per una trattazione completa dell'OSAgent si faccia riferimento alla documentazione VisiBroker.

Un riferimento persistent rimane vivo nell'OSAgent anche al termine del processo server (può essere verificato utilizzando il tool osfind). Un riferimento transient è invece rigidamente associato al ciclo di vita del server e non può avvalersi dei meccanismi di load-balancing e fault-tolerance forniti dall'OSAgent.

Nell'esempio, l'unico oggetto con riferimento persistent è la Factory. I vari carrelli hanno riferimenti transient, sono registrati solo dalla chiamata obj_is_ready della Factory e sono quindi accessibili solo tramite questa.

Anche utilizzando il BOA, l'oggetto CORBA deve specializzare la classe <Inter-faccia>ImlBase. Ecco il codice completo della Factory.

```
package server;

import shopping.*;
import java.util.*;

public class ShoppingCartFactoryImpl extends _ShoppingCartFactoryImplBase {

  private Dictionary allCarts = new Hashtable();
```

```
    // N.B. Registro nell'OSAgent
    public ShoppingCartFactoryImpl(String name) {
      super(name);
    }

    public synchronized ShoppingCart getShoppingCart(String userID) {

      // Cerca il carrello assegnato allo userID...
      shopping.ShoppingCart cart = (shopping.ShoppingCart) allCarts.get(userID);

      // se non lo trova...
      if(cart == null) {

        // crea un nuovo carrello...
        cart = new ShoppingCartImpl();

        // Rende l'oggetto disponibile sull'ORB
        // N.B. _boa() è fornito dalla classe
        //  com.inprise.vbroker.CORBA.Object
        _boa().obj_is_ready(cart);

        System.out.println("Created " + userID "'s cart: " + cart);

        // Salva il carrello nel dictionary associandolo allo userID
        allCarts.put(userID, cart);
      }

      // Restituisce il carrello
      return cart;
    }
  }
```

Il client può ottenere un riferimento alla Factory invocando il metodo bind fornito dall'Helper che esegue anche l'opportuno narrow. Il codice completo della classe client è:

```
  package client;

  import shopping.*;
  import org.omg.CORBA.*;

  public class ShoppingCartClient {

    public static void main(String args[]) {

      if (args.length != 3) {
```

```
      System.err.println("Uso corretto: java ShoppingCartClient userId Autore Titolo");
      return;
    }

    // Crea e inizializza l'ORB
    ORB orb = ORB.init(args, null);

    // Localizza l'oggetto Factory
    ShoppingCartFactory factory
    = ShoppingCartFactoryHelper.bind(orb, "ShoppingCartFactory");

    // Ottengo dalla Factory un oggetto ShoppingCart
    ShoppingCart cart = factory.getShoppingCart(args[0]);

    // Aggiungo un libro
    cart.addBook(new Book(args[1], args[2]));

    // Ottengo la lista dei libri e la stampo
    Book[] list = cart.getBookList();
    for(int i=0; i<list.length; i++)
      System.out.println("Autore " + list[i].Author + " - Titolo " + list[i].Title);

  }
}
```

Prima di avviare il server si attivi l'OSAgent (nel caso si lavori in ambiente distribuito è necessario attivare l'OSAgent sia sulla macchina client che sulla macchina server). Fatto questo, per l'esecuzione dell'esempio si compiano i soliti passi. I riferimenti persistent registrati nell'OSAgent sono controllabili usando il tool osfind.

Attivazione automatica con VisiBroker

Si è detto in precedenza che per l'attivazione automatica di un oggetto, l'adapter attiva i necessari processi server utilizzando le informazioni contenute nell'Implementation Repository. In VisiBroker questo meccanismo è fornito dall'*Object Activation Daemon* (OAD).

L'OAD è un repository che mantiene le informazioni sulle classi che un server supporta, sui loro ID e sulle modalità con cui è necessario attivarle. Le informazioni presenti sull'OAD devono essere registrate ed esistono più modalità di registrazione. Poiché l'OAD è un oggetto CORBA, è possibile costruire un'applicazione che si preoccupi di registrare/deregistrare le varie implementazioni utilizzando i metodi definiti dalla sua interfaccia IDL (per maggior dettagli vedere la documentazione VisiBroker).

Nella pratica è più comune registrare le implementazioni da linea di comando, tipicamente con script di registrazione di più oggetti (magari nella sequenza di boot di una macchina). Comunque, indipendentemente dalle modalità di registrazione, non si dovrà scrivere codice

differente rispetto a quanto fatto in precedenza. Si provi dunque a utilizzare l'attivazione mediante OAD sull'esempio precedentemente scritto per il BOA. Attivati gli OSAgent si avvii il processo OAD con il comando

```
oad -VBJprop JDKrenameBug
```

con alcune versioni di VisiBroker non sarà necessario utilizzare il flag.

L'OAD va attivato solo sulla macchina su cui si vuole eseguire l'oggetto servant. A questo punto si proceda alla registrazione dell'implementazione sull'OAD. Il tool da utilizzare è oadutil che permette di registrare un'interfaccia CORBA (flag -i), con un preciso object name (flag -o), indicando il server che sarà utilizzato per l'attivazione (flag -java). È anche possibile specificare l'activation policy con il flag -p. Si esegua quindi il comando

```
oadutil reg -i shopping::ShoppingCartFactory -o ShoppingCartFactory
-java server.ShoppingCartServer -p shared
```

L'implementazione sarà a questo punto registrata, ma il server non sarà ancora attivo; è possibile controllare il contenuto dell'OAD con il comando

```
oadutil list –full
```

che fornirà un output del tipo

```
oadutil list: located 1 record(s)

Implementation #1:
————————————————
repository_id      =  IDL:shopping/ShoppingCartFactory:1.0
object_name        =  ShoppingCartFactory
reference data     =
path_name          =  vbj
activation_policy  =  SHARED_SERVER
args               =  (length=1)[server.ShoppingCartServer; ]
env                =  NONE

     Nothing active for this implementation
```

Il server sarà attivato alla prima invocazione del client e, poiché si è specificata una politica shared, sarà condiviso anche dai client che successivamente effettueranno una request. Per verificarlo si avvii con le solite modalità l'applicazione client. Lo standard output del processo OAD dovrebbe notificare l'effettiva attivazione del server, in ogni caso è possibile verificare il contenuto dell'OAD con il comando oadutil list visto in precedenza.

In un contesto reale l'attivazione con queste modalità semplifica decisamente le attività di gestione e manutenzione risultando preferibile rispetto all'attivazione manuale dei server.

Se si devono attivare molti oggetti, può essere necessario attivare/disattivare in maniera mirata i vari servant; un meccanismo del genere è realizzabile tramite un cosiddetto service activator.

In linea generale un service activator raggruppa *n* oggetti per i quali è in grado di determinare l'attivazione/disattivazione ad ogni request. Per definire le operazioni di activate/deactivate, l'Activator dovrà implementare l'interfaccia com.visigenic.vbro-ker.extension.Activator. Per una trattazione dell'Activator si rimanda alla documentazione VisiBroker.

Per l'utilizzo di OAD valgono le stesse considerazioni viste in precedenza per OSAgent: fornisce un più semplice utilizzo e notevoli possibilità, ma limita la portabilità lato server. Per una completa portabilità lato server è opportuno utilizzare POA.

Portable Object Adapter (POA)

Il POA entra a far parte delle specifiche CORBA nel 1997 e va a sostituire integralmente a livello funzionale le precedenti specifiche BOA.

La scelta di sostituire integralmente le API BOA è legata all'impossibilità di coniugare i complessi sviluppi presi dalle varie implementazioni proprietarie. Poiché l'ORB è pensato per supportare un numero arbitrario di adapter, BOA e POA possono comunque coesistere.

Lo sviluppo del POA prende l'avvio e si basa sulle molteplici esperienze derivate dalle varie implementazioni del BOA (spesso si dice che POA è semplicemente una versione corretta di BOA), è quindi chiaro che molteplici sono i concetti comuni alle due tipologie di adapter.

Anche per POA valgono dunque le distinzioni effettuate sull'attivazione di implementazioni, sulle tipologie di server e la distinzione tra oggetti persistent o transient.

Per ogni implementazione è definibile un servant manager che, invocato dal POA, crea, attiva o disattiva i vari servant on demand. Il meccanismo dei servant manager aiuta il POA nella gestione degli oggetti server-side. Quasi sempre gli ORB forniscono dei servant manager di default che implementano politiche definite.

È comunque possibile definire direttamente il set di politiche applicate al server senza utilizzare un servant manager, ma operando direttamente sul POA. Nel caso in cui si utilizzi un servant manager, sarà suo il compito di associare la request a un preciso servant, attivandolo o creandolo se necessario. Un servant manager implementa una delle due interfacce di callback ServantActivator e ServantLocator. In generale l'Activator si riferisce a oggetti di tipo persistent, mentre il Locator si riferisce a oggetti di tipo transient.

Indipendentemente da quale interfaccia si utilizzi, le operazioni da definire sono due, una per reperire e restituire il servant, l'altra per disattivarlo. Nel caso di ServantActivator le due operazioni di cui sopra sono incarnate ed etherealize, mentre nel caso di ServantLocator sono preinvoke e postinvoke.

Il POA mantiene una mappa (Active Object Map) dei servant attivi in ogni istante. All'interno della mappa i servant sono associati a uno o più Object Id. Un riferimento a un oggetto sul lato client incapsula l'Object Id e il riferimento al POA ed è utilizzato sul lato server da ORB, POA e servant manager per recapitare la request a un preciso servant.

Non esiste una forma standard dell'Object Id che può essere generato dal POA oppure essere assegnato dall'implementazione. In ogni caso l'Object Id deve essere unico nel namespace e quindi nel POA su cui è mantenuto.

La struttura dei POA è ad albero a partire da un RootPOA. Il RootPOA è sempre disponibile e possiede politiche di default. A partire da questo è possibile generare una gerarchia di nodi POA child con politiche differenti. Sul RootPOA sono mantenuti solo riferimenti transient ed è per questo che nella pratica gli oggetti si installano quasi sempre su child POA creati opportunamente.

Un Child POA può essere creato invocando il metodo create_POA sul suo nodo padre. Per definire le politiche del POA creato bisogna invocare il metodo di creazione fornendogli un oggetto di tipo Policy opportunamente inizializzato. Non è possibile modificare successivamente le politiche di un nodo POA.

Ecco la definizione dei metodi che gestiscono il ciclo di vita di un POA.

```
// IDL
module PortableServer {

  //...

  interface POA {

    //...

    // POA creation and destruction

    POA create_POA(in string adapter_name, in POAManager a_POAManager, in CORBA::PolicyList policies)
                                                                    raises (AdapterAlreadyExists, InvalidPolicy);

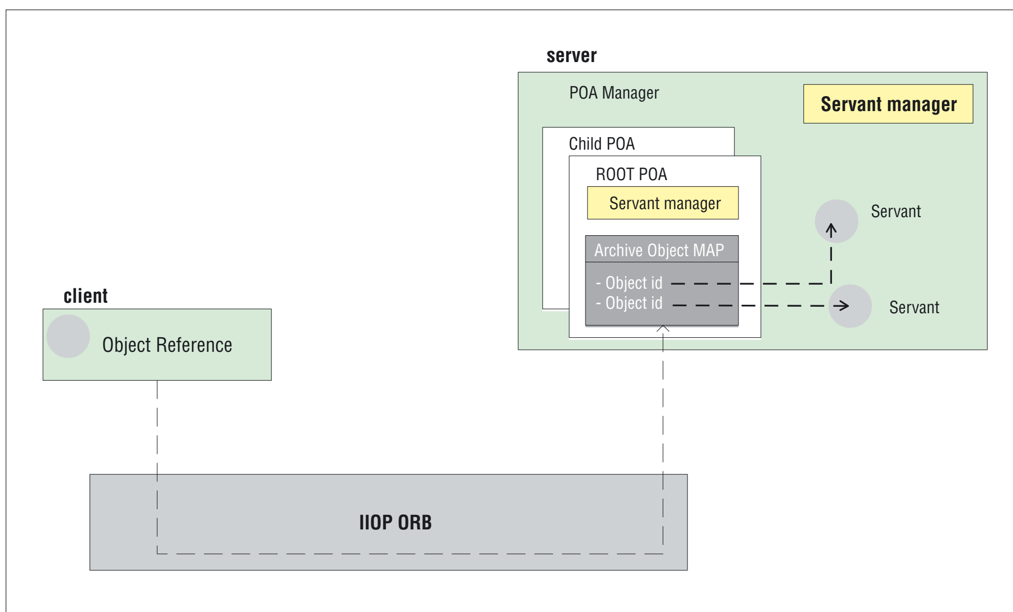
    POA find_POA(in string adapter_name, in boolean activate_it) raises (AdapterNonExistent);

    void destroy(in boolean etherealize_objects, in boolean wait_for_completion);
  };
};
```

L'oggetto Policy va a coprire vari aspetti del runtime environment di un servant associato a un preciso POA. Le specifiche CORBA individuano sette differenti aspetti (in corsivo sono indicati i default):

- Thread: specifica le modalità di trattamento dei thread ovvero singolo thread (SINGLE_THREAD_MODEL) o multithread (ORB_CTRL_MODEL).
- Lifespan: specifica il modello di persistenza (PERSISTENT o TRANSIENT).

Figura 7.8 – *Funzionamento di POA*



- Object Id Uniqueness: specifica se l'Id di un servant deve essere unico (UNIQUE_ID) o può essere multiplo (MULTIPLE_ID).

- Id Assignment: specifica se l'Id deve essere assegnato dall'applicazione (USER_ID) o dal POA (SYSTEM_ID).

- Servant Retention: specifica se il POA mantiene i servant nell'Active Object Map (RETAIN) o si affida a un servant manager (NON_RETAIN).

- Activation: specifica se il POA supporta l'attivazione implicita dei servant (IMPLICIT_ACTIVATION o NO_ IMPLICIT_ACTIVATION).

- Request Processing: specifica come vengono processate le request; usando l'Active Object Map (USE_ACTIVE_OBJECT_MAP_ONLY, da usare con RETAIN) o un servant manager (USE_DEFAULT_SERVANT o USE_SERVANT_MANAGER, da usare con NON_RETAIN).

Per ognuna di queste categorie il POA offre una Factory del tipo create_XXX_policy (create_thread_policy, create_lifespan_policy, …); la costruzione di un oggetto di tipo Policy avviene utilizzando la Factory opportuna.

La scelta della combinazione da adottare è legata alla tipologia di applicazione che si sta realizzando. Ad esempio, una combinazione di RETAIN e USE_ACTIVE_OB-JECT_MAP_ONLY (il default) può essere accettabile per applicazioni che gestiscono un numero finito di oggetti attivati all'avvio (come un Application Server che fornisca servizi continuativamente), ma è una soluzione troppo limitata per un'applicazione che gestisca un gran numero di oggetti. Per applicazioni di questo tipo sono più opportune soluzioni RETAIN che utilizzino servant manager. USE_SERVANT_MANAGER è più indicato per un gran numero di oggetti persistent, mentre USE_DEFAULT_SERVANT per un gran numero di oggetti transient.

Esistono tre differenti tipologie di attivazione di un oggetto mediante POA: attivazione esplicita, attivazione on demand utilizzando un servant manager, e attivazione implicita. L'attivazione esplicita avviene con l'invocazione di alcuni metodi forniti dal POA.

```
// IDL
module PortableServer {
  //...
  interface POA {
    //...
    // object activation and deactivation

    ObjectId activate_object(in Servant p_servant) raises (ServantAlreadyActive, WrongPolicy);

    void activate_object_with_id(in ObjectId id, in Servant p_servant) raises (
                                              ServantAlreadyActive, ObjectAlreadyActive, WrongPolicy);

    void deactivate_object(in ObjectId oid) raises (ObjectNotActive, WrongPolicy);

  };
};
```

Nel caso in cui si usi la registrazione esplicita, il server crea tutti i servant e li registra con uno dei due metodi activate_object.

Nel caso di attivazione on demand, il server si limita a informare il POA su quale servant manager utilizzare per l'attivazione degli oggetti. I metodi per la gestione dei servant manager sono

```
// IDL
module PortableServer {

  //...

  interface POA {

    //...
```

```
      // Servant Manager registration

      ServantManager get_servant_manager() raises (WrongPolicy);

      void set_servant_manager(in ServantManager imgr) raises (WrongPolicy);

    };
  };
```

Si ha un'attivazione implicita effettuando su di un servant inattivo, senza Object Id, operazioni che implichino la presenza di un Object Id nell'Active Map. È possibile solo per la combinazione IMPLICIT_ACTIVATION, SYSTEM_ID, RETAIN. Le operazioni che generano un'attivazione implicita sono

```
  // IDL
  module PortableServer {

    //...

    interface POA {

      //...

      // Identity mapping operations

      ObjectId servant_to_id(in Servant p_servant) raises (ServantNotActive, WrongPolicy);

      Object servant_to_reference(in Servant p_servant) raises (ServantNotActive, WrongPolicy);

    };
  };
```

Anche il POA può essere attivato e disattivato; queste operazioni possono essere effettuate utilizzando il POAManager che è un oggetto associato a uno o più POA. Il POAManager permette anche di bloccare e scartare le request in arrivo.

```
  // IDL
  module PortableServer {

    //...

    // POAManager interface
    interface POAManager {
```

```
    exception AdapterInactive {};

    enum State {HOLDING, ACTIVE, DISCARDING, INACTIVE};

    void activate() raises(AdapterInactive);

    void hold_requests(in boolean wait_for_completion) raises(AdapterInactive);

    void discard_requests(in boolean wait_for_completion) raises(AdapterInactive);

    void deactivate(in boolean etherealize_objects, in boolean wait_for_completion) raises(AdapterInactive);

    State get_state();
  };
};
```

POA in pratica

L'utilizzo di POA permette di configurare l'environment e il comportamento del servant in maniera indipendente dall'ORB. Sarà presentato il solito esempio dello Shopping Cart avendo come punto di riferimento VisiBroker. L'esempio sarà comunque utilizzabile con qualunque altro ORB dotato di POA. Il modo più semplice di utilizzare POA è quello di adottare l'attivazione esplicita senza definire servant manager. Sarà quindi realizzato un server che attivi il servant ShoppingCartFactory in modo persistent. La Factory sarà un servizio sempre disponibile, mentre i singoli carrelli, come già nella versione BOA, saranno oggetti transient. Il primo passo da compiere è quello di ottenere un riferimento al RootPOA.

```
POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
```

Fatto questo è possibile creare un POA con le necessarie politiche. Non è possibile utilizzare direttamente il RootPOA in quanto non supporta la modalità persistent. Un POA di default utilizza modalità multithread, riferimenti transient, Active Object Map. Nell'esempio è quindi sufficiente modificare la proprietà Lifespan da TRANSIENT a PERSISTENT e, con queste politiche, creare il nuovo nodo POA assegnandogli un nome identificativo.

```
org.omg.CORBA.Policy[] policies = {
  rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
};

POA myPOA = rootPOA.create_POA("shopping_cart_poa", rootPOA.the_POAManager(), policies);
```

A questo punto è possibile istanziare il servant e attivarlo sul POA. Poiché l'Id deve essere conosciuto dal client per l'invocazione, in uno scenario semplice è conveniente definirlo esplicitamente nel server.

```
byte[] factoryId = “ShoppingCartFactory”.getBytes();

myPOA.activate_object_with_id(factoryId, factory);
```

Il servant non sarà in condizione di rispondere fino all’attivazione del POA che lo ospita. L’attivazione è effettuata utilizzando il POAManager.

```
rootPOA.the_POAManager().activate();
```

Il ciclo di attesa infinita del server può essere implementato con un metodo dell’ORB (non più impl_is_ready).

```
orb.run();
```

Ecco il codice completo della classe server.

```
package server;

import shopping.*;
import org.omg.PortableServer.*;

public class ShoppingCartServer {

  public static void main(String[] args) {

  try {

    // Inizializza l’ORB.
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);

    // Prende il reference al the root POA
    POA rootPOA = POAHelper.narrow(orb.resolve_initial_references(“RootPOA”));

    // Crea le policies per il persistent POA
    org.omg.CORBA.Policy[] policies = {
      rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
    };

    // Crea myPOA con le date policies

POA myPOA = rootPOA.create_POA(“shopping_cart_poa”, rootPOA.the_POAManager(), policies);

    // Crea l’oggetto Factory
```

```
        ShoppingCartFactoryImpl factory = new ShoppingCartFactoryImpl();

        // Stabilsco l'ID del servant
        byte[] factoryId = "ShoppingCartFactory".getBytes();

        // Attiva il servant su myPOA con il dato Id
        myPOA.activate_object_with_id(factoryId, factory);

        // Attiva il POA
        rootPOA.the_POAManager().activate();

        System.out.println(myPOA.servant_to_reference(factory) + " is ready.");

        // Si mette in attesa delle requests
        orb.run();

      }
      catch (Exception e) {
        e.printStackTrace();
      }
      }
    }
```

Utilizzando POA un servant non deve più specializzare _<NomeInter-faccia>ImplBase, bensì <NomeInterfaccia>POA. La ShoppingCartFactoryImpl sarà quindi

```
    package server;

    import org.omg.PortableServer.*;
    import shopping.*;
    import java.util.*;

    public class ShoppingCartFactoryImpl extends ShoppingCartFactoryPOA {

      private Dictionary allCarts = new Hashtable();

      public synchronized ShoppingCart getShoppingCart(String userID) {

      // Cerca il carrello assegnato allo userID...
      shopping.ShoppingCart cart = (shopping.ShoppingCart) allCarts.get(userID);

      // se non lo trova...
      if(cart == null) {
```

```
        // crea un nuovo carrello...
        ShoppingCartImpl cartServant = new ShoppingCartImpl();

        try {

          // Attiva l'oggetto sul default POA che
          // è il root POA di questo servant
          cart = shopping.ShoppingCartHelper.narrow( _default_POA().servant_to_reference(cartServant));

        } catch (Exception e) {
        e.printStackTrace();
        }

        System.out.println("Created " + userID + "'s cart: " + cart);

        // Salva il carrello associandolo allo userID
        allCarts.put(userID, cart);
        }

        // Restituisce il carrello
        return cart;
      }
    }
```

Si noti che l'attivazione del singolo ShoppingCart è implicita. La generazione dell'Id e la sua registrazione nell'Active Objec Map saranno in questo caso a carico del POA.

L'oggetto ShoppingCartImpl dovrà specializzare ShoppingCartPOA; il codice rimarrà inalterato rispetto all'esempio BOA.

Il client accederà alla Factory in modo molto simile a quanto visto nel BOA. Sul bind andrà specificato l'opportuno Id e il nome del POA su cui è registrato il servant.

```
byte[] factoryId = "ShoppingCartFactory".getBytes();

ShoppingCartFactory factory = ShoppingCartFactoryHelper.bind (orb, "/shopping_cart_poa", factoryId);
```

Ecco il codice completo della classe client.

```
package client;

import shopping.*;
import org.omg.CORBA.*;

public class ShoppingCartClient {
```

```
    public static void main(String args[]) {

      if (args.length != 3) {
        System.err.println("Uso corretto: java ShoppingCartClient userId Autore Titolo");
        return;
      }

      // Crea e inizializza l'ORB
      ORB orb = ORB.init(args, null);

      // ID del servant
      byte[] factoryId = "ShoppingCartFactory".getBytes();

      // Localizza l'oggetto Factory
      // Devo usare il POA name e l'Id del servant
      ShoppingCartFactory factory = ShoppingCartFactoryHelper.bind(orb, "/shopping_cart_poa", factoryId);

      // Ottengo dalla Factory un oggetto ShoppingCart
      ShoppingCart cart = factory.getShoppingCart(args[0]);

      // Aggiungo un libro
      cart.addBook(new Book(args[1], args[2]));

      // Ottengo la lista dei libri e la stampo
      Book[] list = cart.getBookList();
      for(int i=0; i<list.length; i++)
      System.out.println("Autore " + list[i].Author + " - Titolo " + list[i].Title);

    }
  }
```

Parametri per valore

Nel progettare applicazioni distribuite, tipicamente si ragiona suddividendo il dominio dell'applicazione in layer. Il modello più celebre individua tre livelli: presentation, business logic e data/resources. I differenti livelli comunicano attraverso un canale, ma non devono condividere alcun oggetto a livello d'implementazione. Anche per questa ragione middleware come RMI o CORBA operano una netta separazione tra interfaccia e implementazione.

A titolo d'esempio, si immagini un client che si occupa solamente di presentation e un server (magari su un'altra macchina) che gestisce la business logic. Le due applicazioni dovranno al più condividere oggetti che incapsulino dati, ma non dovranno condividere alcun metodo (un metodo di presentation è necessario sul client, un metodo di business è necessario

sul server). Per questi scopi sono quindi sufficienti le strutture dati fornite da IDL (struct, enum, …).

Nonostante queste considerazioni, è spesso utile ricevere/restituire oggetti per valore. In molti casi può essere comodo avere metodi utilizzabili localmente al client e al server.

Fino alle specifiche 2.3 non era possibile il passaggio per valore con oggetti CORBA; come si è visto finora le interfacce sono sempre trattate per riferimento. A differenza di RMI, i metodi non potevano quindi restituire/ricevere via serializzazione oggetti condivisi da client e server. L'introduzione del concetto di valuetype ha ovviato a questa mancanza. Data la recente introduzione del valuetype, alcuni ORB, tra cui quello del JDK 1.2, non supportano ancora questa specifica. Per questa ragione nelle sezioni successive si studieranno il valuetype e alcuni approcci alternativi.

Una possibile soluzione

La prima soluzione è molto semplice e in realtà non è un vero passaggio per copia. L'idea è quella di avere una classe locale (sul server o sul client) che fasci la struttura definita nell'IDL. Questa classe dovrà avere i metodi da utilizzare localmente e, per comodità, un costruttore che riceva in input la struttura dati. Si provi quindi a implementare una semplice (quanto fasulla) funzionalità di login.

```
// IDL
module authentication {

  struct User {
    string userId;
  };

  exception InvalidUserException{};

  interface UserLogin {
    User login(in string user, in string pwd) raises (InvalidUserException);
  };

};
```

Per semplicità si supponga di impostare, sul client e sul server, userId e password da linea di comando (in un contesto reale i dati utente risiederebbero su repository quali basi dati, files o directory server). Il server provvederà a impostare i valori di userId e password sul servant

```
UserLoginImpl login = new UserLoginImpl(args[0], args[1]);
```

Il server effettua le stesse procedure di attivazione e registrazione via Naming Service viste in precedenza. Il codice completo della classe server non viene mostrato.

Ecco invece il codice completo della classe servant.

```
package server;

import authentication.*;

public class UserLoginImpl extends _UserLoginImplBase {

   private String userId;
   private String pwd;

   public UserLoginImpl(String u, String p) {
      super();

      userId = u;
      pwd = p;
   }

   // Metodo di Login
   public User login(String u, String p) throws InvalidUserException {

      if (userId.equals(u) && pwd.equals(p))
         return new User(u);
      else
         throw new InvalidUserException();

   }
}
```

È possibile ora definire il wrapper dello User client-side.

```
package client;

import authentication.*;

public class UserLocalWithMethods {

   private User user;

   public UserLocalWithMethods(User user) {
      this.user = user;
```

```
    }

    // Metodo della classe locale che accede alla struct User
    public String getUserId() {
      return user.userId;
    }

    // Override del metodo toString
    public String toString() {
      return "#User : " + user.userId;
    }
}
```

L'oggetto wrapper sarà creato incapsulando la classe User (che rappresenta la struct IDL). Sull'oggetto sarà possibile invocare i metodi definiti localmente (nel caso in esame getUserId e toString).

```
UserLogin uLogin = UserLoginHelper.narrow(ncRef.resolve(path));

// Effettuo il login e creo l'oggetto wrapper
UserLocalWithMethods user = new UserLocalWithMethods(uLogin.login(args[0], args[1]));

// Utilizzo il metodo della classe locale
System.out.println("Login UserId: " + user.getUserId());

// Utilizzo il metodo toString della classe locale
System.out.println(user);
```

Il client e il server non condivideranno l'implementazione di alcun metodo, ma si limiteranno a condividere la rappresentazione della struttura IDL. La classe con i metodi andrà distribuita semplicemente sul layer logicamente destinato alla sua esecuzione. Potranno esistere wrapper differenti per layer differenti.

Questa soluzione, pur non essendo un effettivo passaggio per valore, rappresenta l'implementazione formalmente più corretta e più vicina allo spirito originale CORBA.

Serializzazione in CORBA

La seconda soluzione utilizza la serializzazione Java e quindi, non essendo portabile, non è molto in linea con lo spirito CORBA. Nel caso però in cui si affronti uno scenario che prevede Java sui client e sui server è comunque una soluzione comoda, simile nell'approccio a RMI. Si ridefinisca l'IDL vista in precedenza

```
module authentication {
```

```
    typedef sequence <octet> User;

    exception InvalidUserException{};

    interface UserLogin {
        User login(in string user, in string pwd) raises (InvalidUserException);
    };

};
```

In questo modo il tipo User, definito come sequence di octet, sarà effettivamente tradotto in Java come array di byte. Il metodo login potrà quindi restituire qualunque oggetto serializzato. L'oggetto condiviso da client e server dovrà essere serializzabile

```
package client;

import java.io.*;

public class User implements Serializable {

    private String userId;

    public User(String userId) {
        this.userId = userId;
    }

    public String getUserId() {
        return userId;
    }

    // Override del metodo toString
    public String toString() {
        return "#User : " + userId;
    }
}
```

L'effettiva serializzazione sarà operata dal metodo login del servant modificato come di seguito riportato.

```
public byte[] login(String u, String p) throws InvalidUserException {

    if (userId.equals(u) && pwd.equals(p)) {
```

```
      // Serializza un oggetto user in un array di byte
      byte[] b = serializza(new client.User(u));
      return b;

   } else
      throw new InvalidUserException();

}
```

Il metodo utilizzato per ottenere l'array di byte serializza un generico oggetto

```
public byte[] serializza(java.lang.Object obj) {

   ByteArrayOutputStream bOut = null;

   try {

      bOut = new ByteArrayOutputStream();
      ObjectOutputStream out = new ObjectOutputStream(bOut);
      out.writeObject(obj);

   } catch(Exception e) {
      e.printStackTrace();
   }

   return bOut.toByteArray();
}
```

Il client opererà in maniera speculare

```
UserLogin uLogin = UserLoginHelper.narrow(ncRef.resolve(path));

// Effettuo il login
byte[] b = uLogin.login(userId, pwd);

// Ottengo l'oggetto User serializzato
User user = (User) deserializza(b);

// Utilizzo il metodo della classe serializzata
System.out.println("Login UserId: " + user.getUserId());

// Utilizzo il metodo toString della classe serializzata
```

```
System.out.println(user);
```

Il metodo deserializza è definito come segue:

```
public java.lang.Object deserializza(byte[] b) {

  java.lang.Object obj = null;

  try {

    ByteArrayInputStream bIn = new ByteArrayInputStream(b);
    ObjectInputStream oIn = new ObjectInputStream(bIn);
    obj = oIn.readObject();

  } catch(Exception e) {
    e.printStackTrace();
  }

  return obj;
}
```

Tale soluzione mina l'interoperabiltà con altri linguaggi e inoltre, dal momento che tratta i parametri come array di byte e non come tipi, diminuisce il livello espressivo di un'interfaccia.

Valuetype

Le soluzioni precedenti sono semplici approcci applicativi; le specifiche CORBA 2.3 hanno definito un approccio standard al passaggio di oggetti per valore. Questa parte di specifiche riveste una grandissima importanza in quanto è uno degli elementi chiave di avvicinamento tra RMI e CORBA. È uno dei meccanismi fondamentali per i tool che implementano RMI over IIOP (si veda più avanti, per una discussione più approfondita).

L'idea chiave che sta alla base delle specifiche CORBA *Object-by-Value* (OBV) è quella di fornire una sorta di serializzazione multipiattaforma.

La definizione di un oggetto serializzabile può essere suddivisa in stato e implementazione. La componente stato è sostanzialmente riconducibile ai valori che hanno gli attributi ed è quindi legata alla singola istanza (escludendo ovviamente attributi static), la seconda componente è l'implementazione dei metodi ed è comune a tutte le istanze.

Anche in Java la serializzazione si limita a rendere persistente lo stato. In fase di deserializzazione l'oggetto viene ricostruito utilizzando la sua definizione (la classe) a partire dal suo stato.

Per la natura delle specifiche CORBA, la definizione di un meccanismo simile a quello descritto comporta un'estensione del linguaggio IDL. La keyword valuetype consente di specificare un nuovo tipo che utilizza il passaggio per valore.

Si modifichi l'IDL vista in precedenza definendo l'oggetto User come valuetype.

```
// IDL
module authentication {

  valuetype User {
    // Metodi locali
    string getUserId();

    // Stato
    private string userId;
  };

  exception InvalidUserException{};

  interface UserLogin {
    User login(in string user, in string pwd) raises (InvalidUserException);
  };

};
```

La definizione mediante valuetype consente di specificare gli attributi con gli opportuni modificatori di accesso (private e public) e le signature dei metodi definite con le stesse modalità adottate nelle interface.

Compilando l'IDL si ottiene la seguente definizione del tipo User (l'esempio non è utilizzabile con il JDK 1.2).

```
package authentication;

public abstract class User implements org.omg.CORBA.portable.StreamableValue {

  protected java.lang.String userId;

  abstract public java.lang.String getUserId ();

  //...

}
```

È necessario fornire una versione concreta dell'oggetto User a partire dalla classe astratta ottenuta dalla precompilazione. Si definisca allora UserImpl come segue:

```
package authentication;
```

```
public class UserImpl extends User {

  public UserImpl() {
  }

  public UserImpl(String userId) {
    this.userId = userId;
  }

  public String getUserId() {
    return userId;
  }

  public String toString() {
    return "#User : " + userId;
  }
}
```

Quando l'ORB riceve un valuetype deve effettuare l'unmarshalling e quindi creare una nuova istanza dell'oggetto opportunamente valorizzata; per fare questo utilizza la Factory associata al tipo in questione. Una factory di default viene creata per ogni valuetype dal processo di compilazione dell'IDL. Nel caso in esame sarà generata una classe UserDefaultFactory.

La classe generata può essere utilizzata come base per la definizione di Factory complesse o semplicemente modificata per ottenere il comportamento voluto; in ogni caso l'ORB deve conoscere l'associazione valuetype–factory. L'associazione può essere stabilita esplicitamente utilizzando il metodo register_value_factory dell'ORB oppure implicitamente utilizzando le naming convention che stabiliscono che, nel caso in cui non esista un'associazione esplicita, l'ORB utilizzi la classe <valuetype>Default-Factory.

Per semplicità si adotti il secondo approccio. Nel caso si utilizzi idlj di Sun la UserDefaultFactory generata non necessita di modifiche. Invece, nel caso si utilizzi VisiBroker, la classe generata dall'IDL sarà incompleta e dovrebbe presentare il codice seguente

```
package authentication;

public class UserDefaultFactory implements org.omg.CORBA.portable.ValueFactory {

  public java.io.Serializable read_value(org.omg.CORBA_2_3.portable.InputStream is) {

    // INSTANTIATE IMPLEMENTATION CLASS ON THE LINE BELOW
    java.io.Serializable val = null;

    // REMOVE THE LINE BELOW AFTER FINISHING IMPLEMENTATION
    throw new org.omg.CORBA.NO_IMPLEMENT();
```

```
        return is.read_value(val);
    }
}
```

Con una versione di JDK differente dalla 1.2 il codice generato potrebbe essere diverso e presentare problemi di compilazione. I commenti generati da idl2java indicano quali modifiche effettuare. Per il caso in esame la Factory può limitarsi a istanziare UserImpl.

```
package authentication;

public class UserDefaultFactory implements org.omg.CORBA.portable.ValueFactory {

    public java.io.Serializable read_value(org.omg.CORBA_2_3.portable.InputStream is) {
        java.io.Serializable val = new UserImpl();
        return is.read_value(val);
    }
}
```

La classe Factory dovrà comunque essere presente nell'ambiente destinato all'unmarshalling (nell'esempio sul client). La restituzione dell'oggetto sarà effettuata dal metodo login di UserLoginImpl modificato come segue

```
public User login(String u, String p) throws InvalidUserException {

    if (userId.equals(u) && pwd.equals(p))
        return new UserImpl(u);
    else
        throw new InvalidUserException();

}
```

Dal punto di vista del client il meccanismo valuetype è invece assolutamente trasparente.

```
User user = uLogin.login(args[0], args[1]);

System.out.println("Login UserId: " + user.getUserId());
```

Nel caso in cui l'ORB non riesca a individuare un'implementazione per il tipo ricevuto, potrà provare a scaricare la corretta implementazione dal chiamante (la funzionalità è simile a quella del codebase RMI). Questa funzionalità può essere disabilitata per ragioni di security.

Come già detto, l'implementazione del meccanismo object-by-value in CORBA ha grande importanza perché consente un semplice utilizzo di RMI over IIOP. È quindi significativo per le specifiche EJB 1.1 che hanno indicato come standard l'adozione di RMI over IIOP. Limi-

tandosi invece allo scenario programmazione distribuita CORBA, valuetype è importante poiché introduce una sorta di serializzazione multilinguaggio e la semantica null value.

CORBA runtime information

In Java esiste la possibilità di ottenere a runtime una serie d'informazioni sulla composizione di un qualunque oggetto (costruttori, metodi, attributi, ecc.). Grazie a queste informazioni è possibile utilizzare un'istanza (invocare metodi, impostare attributi, ecc.) pur non conoscendo a priori la sua classe. Questa potenzialità consente di realizzare soluzioni molto eleganti e flessibili a problemi storicamente complessi quali la definizione/manipolazione di componenti.

CORBA fornisce meccanismi simili a quelli appena descritti e tutti i CORBA object possono fornire a runtime informazioni sulla propria struttura. Nel gergo CORBA queste informazioni sono dette metadata e pervadono l'intero sistema distribuito. Il repository destinato a contenere i metadata è l'Interface Repository che consente di reperire e utilizzare un oggetto ottenendo dinamicamente la sua descrizione IDL.

Queste caratteristiche conferiscono una flessibilità notevole al modello CORBA. Nella pratica risultano particolarmente attraenti per i venditori di tool che sviluppano componenti distribuiti. L'invocazione dinamica, presentando minori performance e maggiori difficoltà di sviluppo, è sicuramente meno interessante per un utilizzo medio.

Introspezione CORBA

Ogni oggetto CORBA specializza `CORBA::Object`; le capacità introspettive di ogni CORBA object derivano da questa superclasse.

```
// IDL
module CORBA {
  //...
  interface Object {
    //...
    IRObject get_interface_def();

    ImplementationDef get_implementation();

    boolean is_nil();

    boolean is_a(in string logical_type_id);

    boolean is_equivalent(in Object other_object);

    boolean non_existent();
```

```
  };
};
```

Il metodo get_interface_def può essere visto come l'equivalente del getClass Java e fornisce i dettagli relativi a un oggetto (attributi, metodi, ecc.). Il metodo restituisce un oggetto generico di tipo IRObject, per ottenere i dettagli di cui sopra è necessario effettuare il narrow a InterfaceDef. Fino alle specifiche 2.3 esisteva un metodo get_interface che restituiva direttamente un oggetto di tipo InterfaceDef: molti ORB lo supportano ancora per backward compatibility. I restanti metodi sono funzioni di test: is_nil è l'equivalente CORBA di "obj == null", is_a controlla la compatibilità dell'oggetto con l'interfaccia data, is_equivalent verifica l'equivalenza tra due reference e non_existent verifica se l'oggetto non è più valido.

Va notato che le signature viste sopra rappresentano le signature IDL del CORBA::Object, ma nella pratica tutti i metodi visti sopra sono forniti dal CORBA.Object con un " _ " davanti (_is_a, _get_implementation, …).

Interface Repository (IR)

L'Interface Repository è il servizio che contiene e fornisce le informazioni sulla struttura IDL degli oggetti. Si è visto nei paragrafi precedenti che, attraverso l'invocazione del metodo get_interface_def, è possibile ottenere un oggetto di tipo InterfaceDef che fornisca tutte le informazioni sulla struttura di una interfaccia.

L'IR contiene in forma persistente gli oggetti InterfaceDef che rappresentano le interfacce IDL registrate presso il repository. L'IR può esser utilizzato anche dall'ORB per effettuare il type-checking nell'invocazione dei metodi, per assistere l'interazione tra differenti implementazioni di ORB o per garantire la correttezza del grafo di derivazione.

Nel caso in cui si intenda utilizzare esplicitamente l'IR sarà necessario registrare l'IDL. La registrazione varia nelle varie implementazioni. VisiBroker prevede due comandi: irep (attiva l'IR) e idl2ir (registra un IDL presso l'IR). Il JDK non fornisce attualmente alcuna implementazione dell'IR.

Interrogando l'IR è possibile ottenere tutte le informazioni descrivibili con un'IDL. La navigazione di queste informazioni avviene tipicamente a partire dalla classe InterfaceDef e coinvolge un complesso insieme di classi che rispecchiano l'intera specifica IDL: ModuleDef, InterfaceDef, OperationDef, ParameterDef, AttributeDef, ConstantDef, …

Dynamic Invocation Interface (DII)

Negli esempi visti finora, per invocare i metodi sugli oggetti CORBA si è utilizzata l'invocazione statica. Questo modello di invocazione richiede la precisa conoscenza, garantita dalla presenza dello stub, dell'interfaccia dell'oggetto. In realtà in CORBA è possibile accedere a un oggetto, scoprire i suoi metodi ed eventualmente invocarli, senza avere alcuno stub precompilato e senza conoscere a priori l'interfaccia esposta dall'oggetto CORBA.

Questo implica la possibilità di scoprire le informazioni di interfaccia a runtime. Ciò è possibile utilizzando una delle più note caratteristiche CORBA: la *Dynamic Invocation Interface*

(DII). La DII opera soltanto nei confronti del client; esiste un meccanismo analogo lato server (*Dynamic Skeleton Interface*, DSI) che consente a un ORB di dialogare con un'implementazione senza alcuno skeleton precompilato.

Purtroppo, anche se DII fa parte del core CORBA, le sue funzionalità sono disperse su un gran numero di oggetti. I passi da compiere per invocare dinamicamente un metodo su un oggetto sono:

- ottenere un riferimento all'oggetto: anche con DII per utilizzare un oggetto è necessario ottenere un riferimento valido. Il client otterrà un riferimento generico ma, non avendo classi precompilate, non potrà effettuare un narrow o utilizzare meccanismi quali il bind fornito dall'Helper.

- ottenere l'interfaccia: invocando il metodo get_interface_def sul riferimento si ottiene un IRObject e da questo, con narrow, l'oggetto navigabile di tipo InterfaceDef. Questo oggetto è contenuto nell'IR e consente di ottenere tutte le informazioni di interfaccia. Con i metodi lookup_name e describe di InterfaceDef è possibile reperire un metodo e una sua completa descrizione.

- creare la lista di parametri: per definire la lista di parametri viene usata una struttura dati particolare, NVList (Named Value List). La creazione di una NVList è effettuata o da un metodo dell'ORB (create_operation_list) o da un metodo di Request (arguments). In ogni caso con il metodo add_item è possibile comporre la lista di parametri.

- creare la Request: la Request incapsula tutte le informazioni necessarie all'invocazione di un metodo (nome metodo, lista argomenti e valore di ritorno). Comporre la Request è la parte più pesante e laboriosa della DII. Può essere creata invocando sul reference dell'oggetto il metodo create_request o la versione semplificata _request.

- invocare il metodo: utilizzando Request esistono più modi di invocare il metodo. Il primo modo è di invocarlo in modalità sincrona con invoke. Il secondo modo è quello di invocarlo in modalità asincrona con send_deferred e controllare in un secondo momento la risposta con poll_response o get_response. Il terzo modo è l'invocazione senza response con send_oneway.

Come si può osservare, un'invocazione dinamica può essere effettuata seguendo percorsi differenti. Lo scenario più complesso implica un'interazione con l'Interface Repository (il secondo e il terzo passo dell'elenco precedente) mediante la quale è possibile ottenere l'intera descrizione di un metodo.

Scenari più semplici non prevedono l'interazione con l'IR e sono a esempio adottati quando ci si limita a pilotare l'invocazione dinamica con parametri inviati da script.

Poiché l'invocazione dinamica è un argomento complesso e di uso non comune, verrà proposto un semplice esempio che non utilizzi l'IR, ma sia pilotato da linea di comando.

Si definisca una semplice interfaccia come segue

```
// IDL
module dii {
  interface DynObject {
    string m0();
    string m1(in string p1);
    string m2(in string p1, in string p2);
    string m3(in string p1, in string p2, in string p3);
  };
};
```

Come si è detto in precedenza, l'uso di DII non ha alcuna influenza sul lato server. Il server si limiterà a istanziare l'oggetto e a registrarlo col nome di DynObject sul Naming Service.

L'implementazione dell'oggetto è molto semplice e ha come unico scopo quello di consentire il controllo dei parametri e del metodo invocato

```
package server;

import dii.*;

public class DynObjectImpl extends _DynObjectImplBase {

  public DynObjectImpl() {
    super();
  }

  public String m0() {
    return "Metodo 0 # nessun parametro";
  }

  public String m1(String p1) {
    return "Metodo 1 # " + p1;
  }

  public String m2(String p1, String p2) {
    return "Metodo 2 # " + p1 + " - " + p2;
  }

  public String m3(String p1, String p2, String p3) {
    return "Metodo 3 # " + p1 + " - " + p2 + " - " + p3;
  }
}
```

Il client deve identificare dall'input da linea di comando il metodo da invocare e i suoi eventuali parametri; il primo passo significativo da compiere è l'accesso al Naming Service.

```
//...
org.omg.CORBA.Object obj = ncRef.resolve(path);
```

Poiché stub, skeleton, Helper e Holder saranno distribuiti solo sul server, non è possibile effettuare un narrow.

A questo punto è possibile iniziare a costruire la Request con il metodo più semplice

```
org.omg.CORBA.Request request = obj._request(args[0]);
```

Si noti che il parametro passato args[0] è il nome del metodo che si intende utilizzare, letto da linea di comando. Ora è possibile costruire la lista di argomenti; nel caso in esame la lista è costruita dinamicamente effettuando un parsing dell'array di input.

```
org.omg.CORBA.NVList arguments = request.arguments();

for (int i = 1; i < args.length; i++) {
   org.omg.CORBA.Any par = orb.create_any();
   par.insert_string(args[i]);
   arguments.add_value("p" + 1, par, org.omg.CORBA.ARG_IN.value);
}
```

Ogni valore della NVList è rappresentato da un oggetto di tipo Any; questo è uno speciale tipo CORBA che può incapsulare qualunque altro tipo e ha un'API che fornisce specifiche operazioni di inserimento ed estrazione di valori (nell'esempio si usano insert_string ed extract_string).

Per invocare il metodo è ancora necessario impostare il tipo di ritorno. Per fare questo si utilizza l'interfaccia TypeCode che è in grado di rappresentare qualunque tipo IDL. Le costanti che identificano i TypeCode dei vari tipi IDL sono fornite da TCKind.

```
request.set_return_type(orb.get_primitive_tc(org.omg.CORBA.TCKind.tk_string));
request.invoke();
```

L'invocazione utilizza la normale chiamata sincrona di metodo data da invoke. Terminata l'invocazione è possibile ottenere e visualizzare la stringa di ritorno.

```
org.omg.CORBA.Any method_result = request.return_value();
System.out.println(method_result.extract_string());
```

L'esecuzione dell'esempio è possibile con VisiBroker utilizzando i consueti passi, in particolare si faccia riferimento all'esempio visto in precedenza sul Naming Service. DII fornisce un meccanismo molto elastico e flessibile che prospetta un sistema assolutamente libero in cui tutti gli oggetti interrogano un Trader Service, individuano e utilizzano dinamicamente i servizi di cui necessitano.

Malauguratamente l'invocazione dinamica presenta alcuni limiti che ne condizionano l'utilizzo. In primo luogo l'invocazione dinamica è decisamente più lenta dell'invocazione statica poiché ogni chiamata a metodo implica un gran numero di operazioni remote.

In secondo luogo la costruzione di Request è un compito complesso e richiede uno sforzo supplementare in fase di sviluppo (lo sviluppatore deve implementare i compiti normalmente svolti dallo stub). L'invocazione dinamica è inoltre meno robusta in quanto l'ORB non può effettuare il typechecking prima dell'invocazione, questo può causare anche un crash durante l'unmarshalling.

Callback

Esistono situazioni in cui un client è interessato al verificarsi di un particolare evento sul server (cambiamento di uno stato, occorrenza di un dato errore, …). In un ambiente distribuito, in cui tipicamente non è possibile conoscere a priori il numero di client, soddisfare questa necessità è problematico.

Il meccanismo più semplice implementabile è il polling: il client periodicamente controlla il valore cui è interessato. Questo approccio è poco robusto e implica un notevole spreco di risorse. Approcci migliori sono forniti da meccanismi di notifica asincrona quali CORBA Event Service o sistemi di messaggistica.

Un'ultima possibilità è quella di utilizzare una callback: sarà il server a invocare direttamente un metodo di notifica sul client o su un oggetto ad esso associato. Questo approccio ha notevoli vantaggi computazionali, ma può presentare difficoltà nel caso in cui la topologia di rete veda il client e il server separati da un firewall.

Tralasciando momentaneamente i problemi di rete, l'utilizzo di callback si presta bene a risolvere alcuni classici problemi di programmazione distribuita. A titolo d'esempio s'immagini la costruzione di una semplice chat. I client devono ricevere una notifica nel caso in cui si connetta un nuovo utente, venga inviato un messaggio o un utente lasci la chat. Si può organizzare l'applicazione adottando un meccanismo simile a quello della gestione degli eventi Java: i client sottoscrivono presso il server chat un servizio di notifica.

La soluzione classica per problemi di questo tipo è data dal pattern Observer. Questo pattern è pensato proprio per quelle situazioni in cui un cambiamento di stato su un oggetto (Subject) ha potenziali impatti su un numero imprecisato di altri oggetti (Observers). In queste situazioni solitamente è necessario inviare agli Observers una generica notifica.

Il numero di Observers deve poter variare a runtime e il Subject non deve conoscere quali informazioni sono necessarie al generico Observer. Per questa ragione la formulazione classica del pattern prevede che siano definiti:

- una classe astratta `Subject` che fornisca i metodi `subscribe`, `unsubscribe` e `notify`;
- una classe `ConcreteSubject` che definisca i metodi di accesso alle proprietà interessate;
- una classe `Observer` che fornisca il metodo di ricezione notifica;

- una classe ConcreteObserver che mantenga un riferimento al ConcreteSubject e fornisca il metodo update utilizzato per riallineare il valore delle proprietà.

Quando un Subject cambia stato invia una notifica a tutti i suoi Observer (quelli che hanno invocato su di lui il metodo subscribe) in modo tale che questi possano interrogare il Subject per ottenere le opportune informazioni e riallinearsi al Subject.

Il pattern Observer, anche conosciuto come Publish/Subscribe, è molto utilizzato nei casi in cui è necessario implementare meccanismi " 1 a *n* " di notifica asincrona. Anche il CORBA Event Service adotta il pattern Observer.

Tornando alla chat, l'implementazione del pattern può essere leggermente semplificata facendo transitare direttamente con la notifica il messaggio cui i client sono interessati.

Nell'esempio in esame ogni client si registrerà come Observer presso il server chat (in uno scenario reale probabilmente si definirebbe un oggetto Observer separato). Il server invocherà il metodo di notifica su tutti gli Observer registrati inviando loro il messaggio opportuno. Nello scenario in esame anche l'oggetto Observer, essendo remoto, dovrà essere attivato sull'ORB e definito via IDL.

```
// IDL
module chat {

  // Forward declaration
  interface SimpleChatObserver;

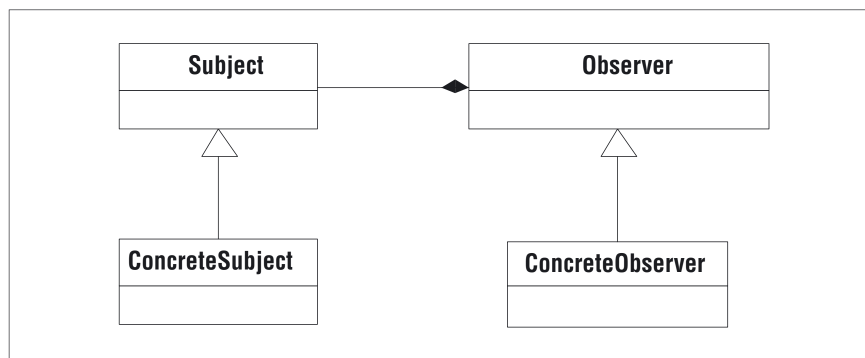
  struct User {
    string userId;
    SimpleChatObserver callObj;
  };

  struct Message {
    User usr;
    string msg;
  };

  // OBSERVER
  interface SimpleChatObserver {
    // N.B. Il server non aspetta alcuna
    // risposta dai vari client
    oneway void callback(in Message msg);
  };

  // SUBJECT
  interface SimpleChat {
    void subscribe(in User usr);
```

Figura 7.9 – *Il pattern Observer*




```
    void unsubscribe(in User usr);
    void sendMessage(in Message msg);
  };

};
```

Nell'esempio non vengono utilizzati gli adapter e l'attivazione viene effettuata mediante connect (quindi è utilizzabile anche con Java IDL). Sarà il client stesso a registrarsi presso l'ORB.

```
package client;

import chat.*;
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SimpleChatClient extends _SimpleChatObserverImplBase {

  private SimpleChat chat = null;
  private User user = null;
  private JTextField tMsg = new JTextField();
  private JButton bSend = new JButton("Send");
  private JTextArea taChat = new JTextArea();

  public SimpleChatClient() {
    super();
```

```
      // qui il codice di inizializzazione UI
   }

   public void init(String userId) throws Exception {

      // Crea e inizializza l'ORB
      ORB orb = ORB.init((String[])null, null);

      // Root naming context
      org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
      NamingContext ncRef = NamingContextHelper.narrow(objRef);

      // Utilizzo il Naming per ottenere il riferimento
      NameComponent nc = new NameComponent("SimpleChat", " ");
      NameComponent path[] = {nc};
      chat = SimpleChatHelper.narrow(ncRef.resolve(path));

      // Si registra presso l'ORB
      // N.B. Il server deve essere in grado di effettuare
      // un'invocazione remota del metodo callback
      orb.connect(this);

      // Crea e registra user
      user = new User(userId, this);
      chat.subscribe(user);
   }

   // Metodo remoto di notifica invocato dal server
   public void callback(Message msg) {
      taChat.append("#" + msg.usr.userId + " - " + msg.msg + "\n");
      tMsg.setText(" ");
   }

   // Lo userId del client è passato da linea di comando
   public static void main(String args[]) throws Exception {
      SimpleChatClient sc = new SimpleChatClient();
      sc.init(args[0]);
   }
}
```

Il riferimento al client viene fatto transitare nell'oggetto User e registrato presso la chat dal metodo subscribe. Per semplicità la deregistrazione del client è associata all'evento di chiusura della finestra (in un contesto reale sarebbe necessario un approccio più robusto).

```
JFrame f = new JFrame("SIMPLE CHAT");
f.addWindowListener(new WindowAdapter() {
  public void windowClosing(WindowEvent e) {
    try {
      chat.unsubscribe(user);
      System.exit(0);
    } catch (Exception ex) {}
  }
});
```

L'oggetto SimpleChatImpl è invece avviato e registrato presso l'ORB da un oggetto server con le modalità consuete. SimpleChatImpl definisce il funzionamento della chat vera e propria.

```
package server;

import chat.*;
import java.util.Hashtable;
import java.util.Enumeration;

public class SimpleChatImpl extends _SimpleChatImplBase {

  Hashtable h = new Hashtable();

  public SimpleChatImpl() {
    super();
  }

  // Aggiunge un utente alla chat
  public synchronized void subscribe(User user) {
    h.put(user.userId, user);

    Message msg = new Message(user, " has joined the chat");
    this.sendMessage(msg);

    System.out.println("Added: " + user.userId);
  }

  // Rimuove un utente dalla chat
  public synchronized void unsubscribe(User user) {
    h.remove(user.userId);

    Message msg = new Message(user, " left the chat");
    this.sendMessage(msg);
```

```
      System.out.println("Removed: " + user.userId);
    }

    // Invia il messaggio a tutti gli utenti
    public void sendMessage(Message msg) {
      User user = null;

      for (Enumeration e = h.elements(); e.hasMoreElements();) {
        user = (User) e.nextElement();

        // Invoca la callback
        try {
          user.callObj.callback(msg);
        } catch (Exception ex) {
          ex.printStackTrace();
        }
      }
    }
  }
```

L'esecuzione dell'esempio segue i consueti passi. Come detto in precedenza, un possibile problema è legato all'eventuale presenza di firewall che inibiscano l'invocazione da server a client.

CORBA e i firewall

Per ragioni di sicurezza tutte le Intranet aziendali utilizzano firewall per limitare il traffico di rete da e verso un certo host. Il firewall agisce come una barriera bidirezionale limitando il traffico sulla base di valutazioni sull'origine/destinatario della richiesta e sul tipo di protocollo utilizzato.

Ogni tecnologia di comunicazione distribuita deve quindi fare i conti con l'eventuale presenza di firewall. Il problema è complesso sia perché non esiste uno standard realmente accettato, sia perché la presenza di un firewall può riguardare tanto il lato server quanto il lato client.

Una tipica configurazione aziendale limita il traffico alla porta 80 riservata al traffico HTTP. Per questa ragione una soluzione comune, adottata anche da RMI, è quella di incapsulare il traffico client/server, JRMP o IIOP, in pacchetti HTTP request/response (HTTP tunneling). In questo modo le richieste/risposte possono attraversare il firewall come le normali attività Web.

Nel caso si utilizzi il tunneling, il Web Server deve essere in grado di riconoscere il particolare tipo di richiesta, estrarre i dati dal pacchetto HTTP e ridirigere la richiesta verso la risorsa opportuna. In questo modo il Web Server agisce sostanzialmente da router.

Per effettuare il routing, il Web Server utilizza CGI o Servlet con ovvi impatti negativi sulle performance. Inoltre, data la natura unidirezionale di HTTP, adottando il tunneling non è possibile utilizzare invocazioni da server a client (notifiche di eventi e callback in genere).

Per queste ragioni il tunneling non si è mai affermato nel mondo CORBA e storicamente ad esso sono sempre state preferite soluzioni alternative proprietarie. Le più celebri soluzioni software fornite dal mercato sono Borland GateKeeper e IONA WonderWall legate agli ORB VisiBroker e OrbixWeb.

Il funzionamento dei due prodotti è similare: ambedue lavorano come Proxy IIOP tra client e server che abbiano restrizioni di security tali da impedire loro di comunicare direttamente. In modo trasparente al client si occupano di ricevere, controllare e "forwardare" il traffico IIOP da e verso l'oggetto remoto opportuno. Tutte le informazioni necessarie al forward transitano con i reference CORBA. Poiché lavorano in ambedue le direzioni, non presentano i limiti della soluzione tunneling sull'utilizzo di callback.

Opportunamente utilizzati questi prodotti consentono anche di utilizzare CORBA con Applet non certificate. Le restrizioni date dalla sandbox non consentono infatti comunicazioni con host differenti da quello da cui l'Applet è stata scaricata e questo è in evidente contrasto con il principio della location transparency CORBA.

Ponendo Gatekeeper o WonderWall sullo stesso host del Web Server, la comunicazione con l'Applet non violerà i limiti della sandbox. A livello TCP l'applet comunicherà direttamente con un oggetto proxy contenuto nei due prodotti. Sarà questo oggetto a effettuare l'invocazione reale al servant CORBA e a gestire in maniera similare la risposta.

Prodotti come quelli citati forniscono soluzioni decisamente superiori al tunneling, ma non appartengono allo standard. CORBA 3.0 ha finalmente fornito specifiche esaustive per i GIOP Proxy Firewall.

Nel caso sia consentito è comunque possibile aprire una porta sul firewall e vincolare le applicazioni CORBA a comunicare soltanto su di essa (le modalità variano da ORB a ORB).

CORBA e J2EE

Tipicamente CORBA viene presentato come middleware alternativo alle altre tecnologie per lo sviluppo di applicazioni distribuite: RMI, EJB, DCOM, ecc.

Poiché però le specifiche danno grande risalto alle necessità di integrazione poste dalle applicazioni Enterprise, CORBA consente in varia misura di interagire con le tecnologie di cui sopra.

Nel caso di DCOM le possibilità di integrazione pongono alcuni problemi e tipicamente si limitano all'utilizzo di bridge che convertono le invocazioni CORBA in opportune invocazioni DCOM.

Completamente differente è invece il ruolo che CORBA si trova a giocare nella piattaforma J2EE. Grazie agli sforzi congiunti di OMG e Sun sono state definite alcune specifiche (object-by value, RMI/IDL, EJB 1.1, CORBA/EJB interop) che non si limitano a portare CORBA a buoni livelli di interoperabilità con la piattaforma Java Enterprise, ma ne fanno un fondamentale elemento infrastrutturale.

La piattaforma J2EE fornisce inoltre funzionalità di chiara ispirazione CORBA. Java Naming e Directory Interface sono correlate al CORBA Naming Service. Java Transaction API e Java Transaction Service sono correlate al CORBA Object Transaction Service. Molte similitudini possono essere inoltre individuate in altri ambiti J2EE: gestione della Security, gestione della persistenza e utilizzo di *Messaging Oriented Middleware* (MOM).

CORBA vs RMI

Come si è avuto modo di osservare esistono notevoli somiglianze tra l'utilizzo di CORBA e quello di RMI. In realtà è evidente che molte delle conclusioni e dei dettami OMG sono confluiti nella definizione dell'architettura RMI.

Ad alto livello l'utilizzo di RMI è più semplice di quello CORBA. La ragione di questo va ricercata nell'implicita semplificazione che si ha dovendo fornire specifiche monolinguaggio. Inoltre RMI può contare su un notevole insieme di funzionalità base offerte dal linguaggio Java e non supportate dagli altri linguaggi utilizzabili con CORBA (si pensi alla serializzazione e all'introspezione).

In generale CORBA può offrire performance superiori a RMI e migliori soluzioni di clustering e load balancing. Le differenze possono essere più o meno significative a seconda dell'ORB utilizzato.

La scelta tra l'utilizzo dell'una o dell'altra tecnologia è quindi legata alle esigenze d'utilizzo. In uno scenario semplicemente Java senza alcuna necessità di integrazione con altri linguaggi la scelta RMI potrà essere accettabile. In uno scenario Enterprise eterogeneo o con significative esigenze di tuning e performance sarà da preferire l'adozione di CORBA.

Sebbene CORBA e RMI vengano spesso presentate come tecnologie alternative, esiste la concreta possibilità di farli cooperare utilizzando RMI con il protocollo CORBA IIOP.

RMI – IIOP

L'utilizzo tipico di RMI prevede l'adozione del protocollo di trasporto proprietario *Java Remote Method Protocol* (JRMP). Nel 1998 specifiche prodotte da Sun e IBM hanno introdotto la possibilità di utilizzare RMI sul protocollo IIOP. L'elemento fondamentale di RMI-IOOP è costituito dalle specifiche Object-by-value che consentono di adottare con CORBA il passaggio di parametri per valore tipico di RMI.

La definizione di RMI-IIOP fornisce allo sviluppatore i vantaggi di semplicità RMI uniti alle caratteristiche di portabilità/interoperabilità del modello CORBA. La soluzione è adottata anche dall'infrastruttura EJB.

Per lo sviluppatore Java si ha inoltre il vantaggio di non dover imparare il linguaggio IDL. La definizione delle interfacce viene operata direttamente in Java con modalità RMI. A partire dalle interfacce Java un compilatore apposito genererà tutte le classi di infrastruttura CORBA.

Con VisiBroker vengono forniti due compilatori: java2iiop si occupa di generare stub, skeleton, Helper e Holder, java2idl consente di ottenere la rappresentazione IDL.

Anche il JDK fornisce un completo supporto per RMI/IIOP. Utilizzando il JDK 1.3 questo è già fornito con l'SDK, mentre nel caso si utilizzi un JDK precedente sarà necessario scaricarlo a parte insieme ai tool RMI/IIOP (si veda [RMI over IIOP] in bibliografia). In ogni caso andrà utilizzata la nuova versione del compilatore rmic con i flag `-iiop` o `-idl`.

Le specifiche EJB 1.1 indicano RMI-IIOP come API standard di comunicazione. L'uso di RMI-IIOP è la chiave della compatibilità tra CORBA ed EJB. Già prima di queste specifiche alcuni Application Server fornivano il layer EJB sopra una infrastruttura CORBA (Borland Application Server, ecc.). Altri produttori utilizzavano invece implementazioni proprietarie dell'API RMI (Bea WebLogic, ecc.).

Bibliografia

[OMG]
La Home Page di OMG, http://www.omg.org

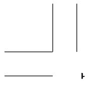
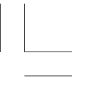
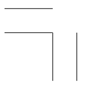
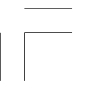
[CORBA]
La specifica, ftp://ftp.omg.org/pub/docs/ptc/96-03-04.pdf

[IDL]
Java IDL, http://java.sun.com/products/jdk/idl

[VIS]
Visibroker, http://www.borland.com/visibroker/

[FREE]
Free CORBA page, http://adams.patriot.net/~tvalesky/freecorba.html

[RMI]
RMI over IIOP, http://java.sun.com/products/rmi-iiop/index.html

Capitolo 8

Enterprise JavaBeans

GIOVANNI PULITI

La specifica EJB

La specifica di Enterprise JavaBeans definisce un modello di tipo CTM (Component Transaction Monitors), uno dei modelli più avanzati nel mondo delle business application.

Da un punto di vista architetturale EJB è semplicemente un modo diverso di vedere e di implementare il modello della computazione distribuita, che ha visto nel corso degli anni numerose variazioni sul tema: una delle più famose è quella basata sugli oggetti distribuiti, i quali possono essere eseguiti in remoto e su più macchine indipendenti fra loro.

Già prima di EJB erano disponibili alcune interessanti e importanti alternative: da un lato Remote Method Invocation (RMI) permette in maniera piuttosto semplice di realizzare applicazioni distribuite e rappresenta la soluzione full-Java più semplice, anche se probabilmente meno flessibile.

In alternativa, CORBA è spesso considerato il sistema di gestione degli oggetti remoti e distribuiti più versatile, eterogeneo, potente, ma anche più complesso. Quando fu realizzata la prima edizione del *Manuale pratico di Java*, CORBA veniva visto ancora come un valido strumento per l'integrazione e lo sviluppo di applicazioni distribuite. Attualmente lo scenario tecnologico vede EJB come attore principale se non unico, specie nella realizzazione di applicazioni nuove. CORBA sta lentamente riducendo la sua diffusione relegato sempre più a contesti legacy o a casi particolari eterogenei.

Ultimamente Microsoft sta spingendo sulla tecnologia .Net, che di fatto rappresenta il tentativo della casa di avvicinarsi al mondo enterprise. Pur essendo valutata una tecnologia interessante e promettente (sebbene molte parti siano direttamente "ispirate" alla piattaforma Java), .Net è ancora carente, soprattutto per la parte riguardante gli application server del tutto assenti. Non si faranno in questo capitolo confronti con le tecnologie alternative, dato che questo richiederebbe uno spazio certamente maggiore quello qui a disposizione.

La differenza fondamentale di EJB rispetto agli altri modelli è rappresentata dal supporto offerto per la realizzazione di applicazioni distribuite: RMI e CORBA infatti sono esclusivamente motori di oggetti distribuiti (solo Java nel primo caso, *language neutral* nel secondo), dove i cosiddetti distributed object services devono essere implementati a mano o forniti per mezzo di software terze parti.

In EJB invece tutto quello che riguarda la gestione dei servizi è disponibile ed utilizzabile in maniera automatica e trasparente agli occhi del client.

Per servizi si intendono quelle funzionalità atte alla implementazione dei tre aspetti qui di seguito descritti.

Transazioni

Sinteticamente una transazione è definibile come un insieme di operazioni che devono essere svolte in maniera atomica, oppure fallire totalmente. La sola atomicità non però è un requisito sufficiente a garantire la correttezza delle operazioni svolte e la coerenza dei dati memorizzati: come si avrà modo di vedere in seguito, per definire il livello di bontà di un sistema transazionale spesso ci si riferisce all'acronimo ACID.

La transazionalità è molto importante nel mondo del commercio elettronico e nella maggior parte delle casistiche reali. I container EJB offrono un sistema automatico per la gestione delle transazioni.

Security

Java è nato come linguaggio estremamente attento al concetto di sicurezza, aspetto ancora più importante quando si ha a che fare con sistemi distribuiti. Nella specifica EJB è stato inserito un supporto integrato alla security basandosi su quello offerto dalla piattaforma Java 2.

Scalabilità

La filosofia con cui EJB è stato progettato, soprattutto la modalità con cui opera, consentono un elevato livello di flessibilità e scalabilità in funzione del traffico dati e del numero di clienti. Ad esempio la politica per la decisione di quali, quanti e come i vari bean vengano forniti ai vari client è totalmente gestita dal server.

Da notare che EJB è costruito sopra il modello di RMI del quale utilizza le funzionalità e la filosofia di base per realizzare strutture distribuite, estendendone come visto i servizi in maniera trasparente.

Architettura

La specifica di EJB definisce una architettura standard per l'implementazione della business logic in applicazioni multi-tier basate su componenti riusabili. Tale architettura è fondata essenzialmente su tre componenti: il server, i container e i vari client.

Server EJB

Il server EJB ha lo scopo di incapsulare tutto quello che sta sotto lo strato EJB (applicazioni legacy, servizi distribuiti) e di fornire ai contenitori gli importanti servizi di base per i vari componenti installati.

Container EJB

Intercettando le invocazioni dei client sui metodi dei bean, il container è in grado di effettuare diversi controlli molto importanti, in modo da offrire alcune funzionalità legate al ciclo di vita dei vari componenti, alla gestione delle transazioni ed alla security management.

Dal punto di vista implementativo, al fine di permettere la corretta comunicazione fra componente e contenitore, chi sviluppa un componente deve implementare alcune interfacce standard.

Client

Il client infine rappresenta l'utilizzatore finale del componente. La visione che ha del componente è resa possibile grazie a due interfacce (`HomeInterface` e `RemoteInterface`), la cui implementazione è effettuata dal container al momento del deploy del bean.

La `HomeInterface` fornisce i metodi relativi alla creazione del bean, mentre l'altra offre al client la possibilità di invocare da remoto i vari metodi di business logic. Implementando queste due interfacce, il container è in grado di intercettare le chiamate provenienti dal client e al contempo di fornire ad esso una visione semplificata del componente. Il client non ha la percezione di questa interazione da parte del container: il vantaggio è quindi quello di offrire in modo indolore la massima flessibilità, scalabilità oltre a una non indifferente semplificazione del lavoro da svolgere.

Tipi di Enterprise JavaBeans

Nella versione 1.1 della specifica di EJB sono stati definiti due tipi di componenti, gli entity bean e i session bean. A partire dalla 2.0 sarà possibile disporre di un terzo tipo di componente, orientato alla gestione messaggi asincroni basati sulla API Java Message Service (JMS).

Un entity bean rappresenta un oggetto fisico, un concetto rappresentabile per mezzo di una parola: per questo motivo spesso rappresentano i soggetti in gioco nella applicazione, anche se generalmente svolgono un ruolo passivo, essendo utilizzati prevalentemente dal client.

Gli entity sono in genere mantenuti persistenti per mezzo del supporto di un database di qualche tipo. A seconda del meccanismo di persistenza messo in atto per sincronizzare i dati del bean con quelli nel database sottostante, si definiscono due tipologie di entity bean: bean con persistenza gestita dal bean stesso (Bean Managed Persistence o BMP) e bean con persistenza gestita dal container (Container Managed Persistence o CMP).

Nel primo caso il programmatore deve solo implementare alcuni metodi di callback invocati dal container, all'interno dei quali si dovranno inserire le istruzioni di scrittura/lettura verso il database (tipicamente stament SQL). Il container sa quando e come invocare tali metodi, il programmatore dice cosa deve essere fatto in concomitanza di tali eventi.

I CMP invece adottano il cosiddetto Abstract Persistence Schema: il programmatore non deve implementare nessuna operazione di accesso ai dati ed anzi tutte le operazioni di creazione, caricamento, salvataggio, ricerca sono definite in vario modo tramite XML. Questo sistema, benché concettualmente molto complesso, permette di realizzare applicazioni in cui tutto lo strato di accesso ai dati e soprattutto di mapping Object-Relational (ORM) sia completamente indipendente dal database sottostante.

I session bean invece possono essere visti come un'estensione della applicazione client: sono responsabili della gestione della applicazione o dei vari task, ma non dicono niente del contesto all'interno del quale tali operazioni sono eseguite. I session bean possono essere di tipo stateless o stateful a seconda che il bean mantenga in memoria lo stato della conversazione fra client e server, ossia che fra due invocazioni successive dei metodi di un bean, le variabili del session mantengano i valori precedentemente impostate.

Prendendo in considerazione il caso dell'organizzazione di una scuola, si può immaginare che l'aula stessa sia rappresentabile con un entity bean (entity bean `StudentClass`), così come gli insegnanti (entity bean `Teacher`), i banchi (entity bean `Book`), gli studenti (entity bean `Student`) i libri e cosi via. Tutto quello che invece riguarda la coordinazione delle varie risorse come l'assegnazione di studenti alle classi, degli insegnanti alle classi, dei banchi e dei libri agli studenti sono operazioni di business logic che in genere devono essere inserite all'interno di un session bean.

Esempio applicativo

Nel procedere di questo capitolo verranno presentati i vari concetti della teoria di EJB facendo di volta in volta riferimento ad una applicazione di esempio che si chiama SchoolManager: il progetto implementa una semplice gestione di una ipotetica scuola. L'obiettivo non è tanto quello di riprodurre un caso fedelmente aderente alla realtà, ma piuttosto quello di mostrare alcune delle funzionalità più importanti di EJB tramite un semplice esempio. SchoolManager non è quindi una applicazione completa in ogni sua parte e anzi molti passaggi possono essere completati dal lettore.

Alcune scelte progettuali di SchoolManager sono state fatte appositamente per verificare il funzionamento di EJB e probabilmente non hanno una corrispondenza fedele agli occhi del lettore. Particolare risalto è dato alla gestione dei dati in CMP 2.0 (di cui si parlerà nella parte finale del capitolo) e quindi alle possibili interazioni/relazioni che si possono instaurare fra oggetti diversi. Ecco le entità che partecipano alla applicazione:

- `Student`: uno studente;
- `Curriculum`: il curriculum dello studente;
- `TestResult`: un documento che rappresenta un risultato di un esame/compito/test;
- `Teacher`: un insegnante;

- StudentClass: una classe di studenti;
- Book: un libro assegnato ad uno studente;
- Desk: il banco dove uno studente si deve sedere;

I legami che si instaurano fra questi componenti sono stati pensati per cercare di coprire le 7 relazioni che sono possibili fra oggetti diversi e che sono riportati qui di seguito

Nel caso specifico gli oggetti sono legati nel seguente modo:

```
Student  (1)  --------> (1)   Desk
Student  (1)  <-------> (1)  Curriculum
Student  (1)  --------> (n)   Book
Student  (n)  --------> (1)   School
Student  (1) <-------> (n)   TestResult
Student  (n) <-------> (1)   StudentClass
Teacher  (n) <-------> (m)  StudentClass
```

Nella parte conclusiva del capitolo (fig. 8.18) è riportata una rappresentazione grafica dello schema relazionale. Una approfondita spiegazione di ogni parte della applicazione richiederebbe un capitolo apposito per cui non verrà qui affrontata: il lettore che fosse interessato ad approfondire i vari aspetti legati all'applicazione, potrà leggere il tutorial che è annesso agli esempi scaricabile dal sito internet di MokaByte (http://www.mokabyte.it).



Nella prima edizione del *Manuale pratico di Java* si era cercato di effettuare una trattazione più ampia possibile che tenesse conto della specifica EJB in versione 1.0, 1.1 e in alcuni casi della 2.0. In questa seconda edizione del libro, si farà riferimento principalmente alle release 1.1 e 2.0, mentre la 1.0 sarà occasionalmente presa in esame per confrontarla con le nuove versioni.

Tabella 8.1 – *Le possibili relazioni fra oggetti in una applicazione*

descrizione	simbolismo
1 a *1* monodirezionale	(1) --------> (1)
1 a *1* bidirezionale	(1) <-------> (1)
1 a *n* monodirezionale	(1) --------> (n)
n a *1* monodirezionale	(n) --------> (1)
1 a *n* bidirezionale	(1) <-------> (n)
m a *n* monodirezionale	(n) --------> (m)
m a *n* bidirezionale	(n) <-------> (m)

Strutturazione dei vari componenti

Per poter realizzare un enterprise bean è necessario implementare due classi ed estendere due interfacce (sia che si tratti di un entity che di un session). Per comodità spesso ci si riferisce al concetto di bean come tutto l'insieme degli oggetti e delle interfacce che compongono il bean: il nome utilizzato è quello che fa riferimento alla bean class. Di seguito se ne riportano le caratteristiche.

Remote interface

Per prima cosa un bean deve definire una interfaccia remota tramite l'estensione della interfaccia javax.ejb.EJBObject, la quale deriva a sua volta dalla ben nota java.rmi.Remote. I metodi esposti della interfaccia remota costituiscono i cosiddetti business method del bean, ossia le funzionalità che il client sarà in grado di utilizzare da remoto. L'oggetto che implementerà tale interfaccia e che viene creato dal server al momento del deploy, viene detto in genere EJBObject.

Home interface

Questa interfaccia definisce il set di metodi necessari per gestire il ciclo di vita del componente: tali metodi sono invocati dal container durante le fasi di creazione ed installazione nel container, durante la rimozione, o in corrispondenza di operazioni di ricerca per mezzo di chiavi univoche. L'interfaccia da estendere in questo caso è la javax.ejb.EJBHome, derivante anch'essa da java.rmi.Remote, ed è per questo motivo che caso l'interfaccia creata viene detta EJBHome o più semplicemente home interface.

Bean class

Per costruire un entity bean va definita una classe derivante da javax.ejb.EntityBean, mentre per i session bean l'interfaccia da estendere è la javax.ejb.SessionBean.

Questa classe contiene la business logic del bean: essa non deve implementare le due interfacce remote appena viste, anche se suoi metodi devono corrispondere a quelli definiti nella remote interface ed in qualche modo sono associabili a quelli specificati nella home interface.

Il fatto che per creare un bean si debbano estendere delle interfacce che poi sono implementate dal container al momento del deploy, è la base della filosofia di EJB: infatti oltre alla notevole semplificazione che questo comporta, tale sistema consente l'interazione fra il client e l'oggetto remoto in modo indiretto, tramite l'intermediazione del container, il quale si preoccupa di creare nuove istanze di oggetti remoti e di verificarne la corretta installazione e il giusto funzionamento, e così via.

Primary key

Normalmente ogni bean, specie nel caso degli entity, è rappresentabile per mezzo dei valori delle variabili in esso contenute Tali proprietà finiscono spesso per diventare la chiave (semplice o composta da più campi) tramite la quale identificare univocamente il bean.

Nel caso in cui il bean sia particolarmente complesso, o i dati non siano utilizzabili direttamente, si può procedere a definire una classe apposita che svolgerà il ruolo di chiave per l'oggetto. In genere questa classe verrà utilizzata dai vari metodi di ricerca, messi a disposizione dalla interfaccia home, come si avrà modo di vedere in seguito.

Implementazione di un entity bean

Al fine di comprendere meglio come strutturare le varie parti di un componente, si riconsideri il caso relativo alla gestione di una scuola, analizzando cosa sia necessario fare per implementare un entity bean; il caso dei session bean non è molto differente.

La struttura base delle classi e delle interfacce necessarie per mettere in piedi lo scheletro di una applicazione di questo tipo è riportata nelle figg. 8.1 e 8.2.

Il codice per la remote interface è il seguente:

```
public interface StudentRemote extends EJBObject {
   public void addBooks(Collection books) throws RemoteException;
   public void assignCurriculum(String curriculumId) throws RemoteException;
   public void addTestResults(Collection testResults) throws RemoteException;
   public void addTestResult(TestResultLocal testResultLocal) throws RemoteException;
   public void addBook(BookLocal BookLocal) throws RemoteException;
   public void setAddress(String address) throws RemoteException;
   public String getAddress() throws RemoteException;
```

Figura 8.1 – *Organizzazione gerarchica di remote e home interface.*

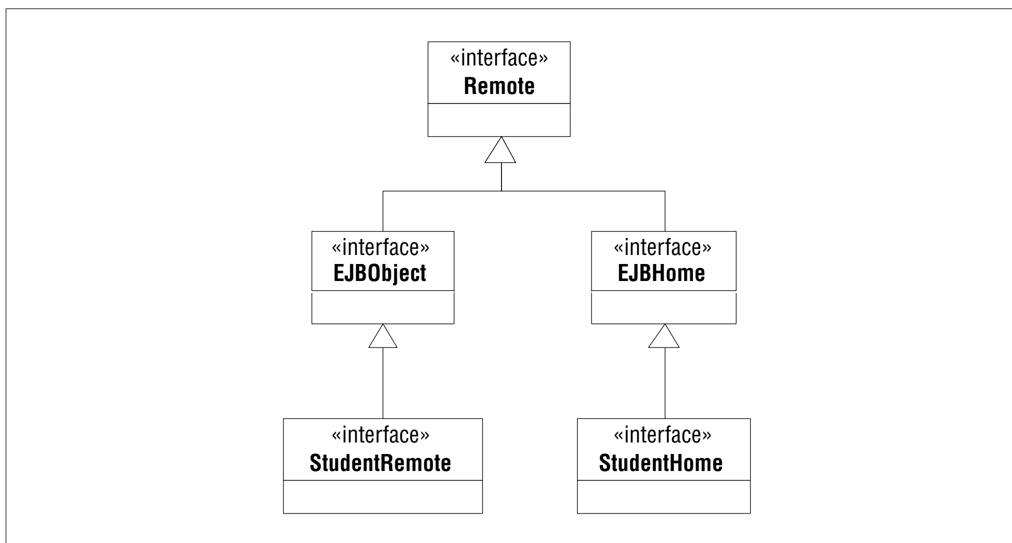
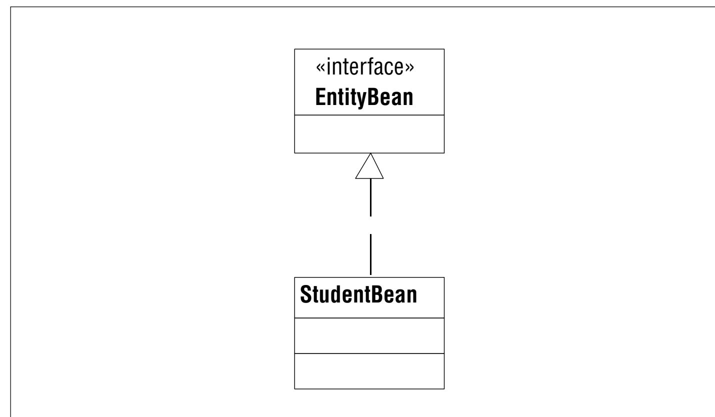


Figura 8.2 – *La classe Student, che implementa il concetto di studente.*




```
    public void setAge(String age) throws RemoteException;
    public String getAge() throws RemoteException;
    public void setBirthday(String birthday) throws RemoteException;
    public String getBirthday() throws RemoteException;
    public void setEmail(String email) throws RemoteException;
    public String getEmail() throws RemoteException;
    public void setFirstName(String firstName) throws RemoteException;
    public String getFirstName() throws RemoteException;
    public void setLastName(String lastName) throws RemoteException;
    public String getLastName() throws RemoteException;
    public void setPassword(String password) throws RemoteException;
    public String getPassword() throws RemoteException;
    public String getId() throws RemoteException;
}
```

Il significato di tali metodi dovrebbe essere piuttosto intuitivo partendo dal nome: i metodi di set/get permettono di accedere ai campi dell'oggetto studente, mentre gli altri eseguono semplici operazioni di assegnazione. La home invece potrebbe essere qualcosa del tipo:

```
public interface StudentHome extends EJBHome {
    public StudentRemote create(StudentDTO dto) throws CreateException, RemoteException;
    public StudentRemote findByPrimaryKey(String id) throws FinderException, RemoteException;
    public Collection findAll() throws FinderException, RemoteException;
    public Collection findByFirstName(String firstName) throws FinderException, RemoteException;
    public Collection findByTeacherId(String teacherId) throws FinderException, RemoteException;
}
```

Nel codice corrispondente alla classe StudentBean, l'implementazione dei metodi del bean varia a seconda che si tratti di un bean con persistenza gestita direttamente (BMP) o dal container (CMP) concetti di cui si parlerà successivamente.

```
public abstract class StudentBean   implements EntityBean
  public void ejbLoad() {
    …
  }
  public void ejbStore() {
    …
  }
  public void ejbActivate() {
    …
  }
  public void ejbPassivate() {
    …
  }
  public void setEntityContext(EntityContext entityContext) {
    this.entityContext = entityContext;
  }

  public void unsetEntityContext() {
    this.entityContext = null;
  }

// segue implementazione dei metodi di accesso ai campi
…
// segue implementazione dei metodi di assegnazione
}
```

Per prima cosa si possono individuare i vari metodi relativi al ciclo di vita del componente, invocati in modalità di callback dal container: essi sono necessari per il modello EJB, ma non fanno parte dell'interfaccia pubblica del componente; inoltre essendo parte integrante della interfaccia EntityBean, quindi delle specifiche del modello EJB, ne è obbligatoria la definizione ma non l'implementazione.

Al loro interno si potranno inserire tutte le operazioni che si desidera siano effettuate durante una determinata fase del ciclo di vita del bean, senza che ci si debba preoccupare di quando e come essi verranno invocati, ossia tralasciando i problemi legati a come e quando verranno effettuati i passaggi di stato sul componente.

Possono però essere pensati come sistema di notifica sul bean di un imminente passaggio di stato nell'ambito del suo ciclo di vita.

L'adozione di questa architettura permette di operare una netta separazione fra il lavoro di chi sviluppa il bean (che non ha né la necessità né la volontà di sapere come sia implementato il

server, potendosi concentrare esclusivamente sui vari business method del componente) e chi sviluppa il server (che non deve conoscere, né oltretutto potrebbe, i dettagli implementativi delle varie applicazioni distribuite basate sul modello EJB).

I metodi ejbCreate() ed ejbPostCreate() sono invocati dal container sul componente rispettivamente subito prima e subito dopo la creazione del componente, mentre il metodo ejbRemove() comunica al componente che sta per essere rimosso dal server, e che i dati relativi, memorizzati nel db, stanno per essere cancellati.

Invece ejbLoad() ed ejbStore() notificano che lo stato del componente sta per essere sincronizzato con il database. In particolare il metodo ejbStore() viene chiamato dal container subito prima che il componente venga memorizzato nel database, in modo da consentire al programmatore di effettuare tutte le operazioni di sincronizzazione e messa a punto prima che il componente stesso sia reso persistente.

Analogamente l'invocazione di ejbLoad() avviene immediatamente dopo la fase di caricamento del componente dal db.

I metodi ejbActivate() e ejbPassivate() notificano al componente che sta per essere attivato o disattivato.

Gli altri metodi presenti nel bean sono i cosiddetti accessori e mutatori (setXXX e getXXX) e permettono di gestire le proprietà dell'oggetto remoto: insieme ai business method sono visibili dal client e rappresentano perciò l'interfaccia pubblica vera e propria del componente.

Per la ricerca di una particolare istanza di bean da parte del client, si deve provvedere alla definizione di almeno un metodo di ricerca per chiave univoca: il findByPrimaryKey() è il metodo che necessariamente deve essere presente nella home interface, ma si possono aggiungere altri metodi che consentano di reperire i componenti tramite altri criteri. In seguito tali concetti saranno ripresi ed approfonditi.

La chiave primaria deve essere definita dallo sviluppatore del bean, e deve essere di un tipo serializzabile. Ad esempio si può pensare di definire una chiave nel seguente modo:

```
public class StudentPK implements Serializable {
  private int id;
}
```

Implementazione di un session bean

Per quanto riguarda la creazione di un session bean, il procedimento, con le dovute modifiche, è sostanzialmente analogo al caso degli entity. Innanzitutto deve essere implementata l'interfaccia javax.ejb.SessionBean invece della EntityBean. Questo comporta la presenza del metodo ejbCreate() ma non di ejbPostCreate().

Dato che un bean di questo tipo non è persistente, non sono presenti i metodi ejbLoad() ed ejbStore(). Similmente è presente il metodo setSessionContext() ma non unsetSessionContext().

Infine un session bean dovrebbe fornire un metodo ejbRemove() da utilizzare per avvertire il bean quando il client non ne ha più bisogno: i questo caso però, differentemente da ciò che accade con un entity, non vi è nessuna rimozione dei dati dal database, dato che i session bean non prevedono nessun meccanismo di memorizzazione persistente.

Sempre per lo stesso motivo i session bean non sono associati ad una chiave primaria e quindi non è presente nessun metodo di ricerca.

EJB dietro le quinte

Non è stato detto niente fino ad ora relativamente al processo si implementazione delle due interfacce remote ed home. Per non lasciare troppo in sospeso il lettore, è forse utile fare una analisi leggermente più approfondita sia su EJBObject che su EJBHome. Tali interfacce sono implementate dal container al momento del deploy, tramite la creazione di un oggetto remoto che funziona come wrapper per il bean stesso (ossia per la classe che implementa ad esempio l'interfaccia EJBEntity).

Il modo con cui tale oggetto viene creato dipende molto dall'implementazione del server, ma si basa sulle informazioni contenute nel bean e nei vari descriptor file: esso lavora in stretto contatto con il container e provvede a fornire tutte le funzionalità di gestione delle transazioni, di security ed altre funzionalità a livello di sistema.

La configurazione più utilizzata dal server è quella in cui l'oggetto remoto implementa l'interfaccia remota e contiene un riferimento alla classe che implementa l'entity interface.

Figura 8.3 – *Architettura di organizzazione delle classi remote e del bean secondo la configurazione tipica a wrapper.*

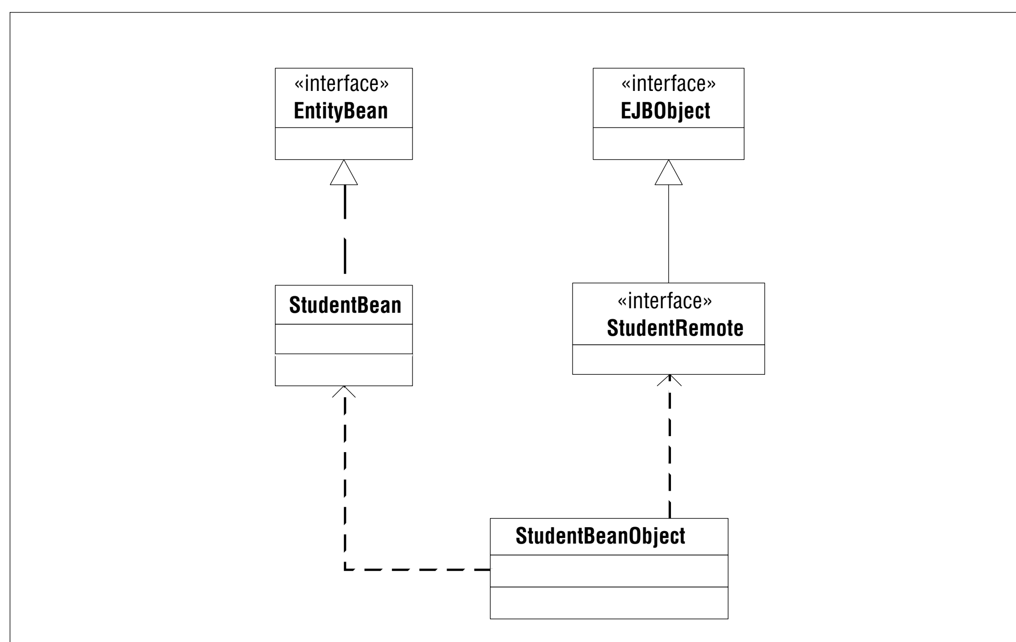
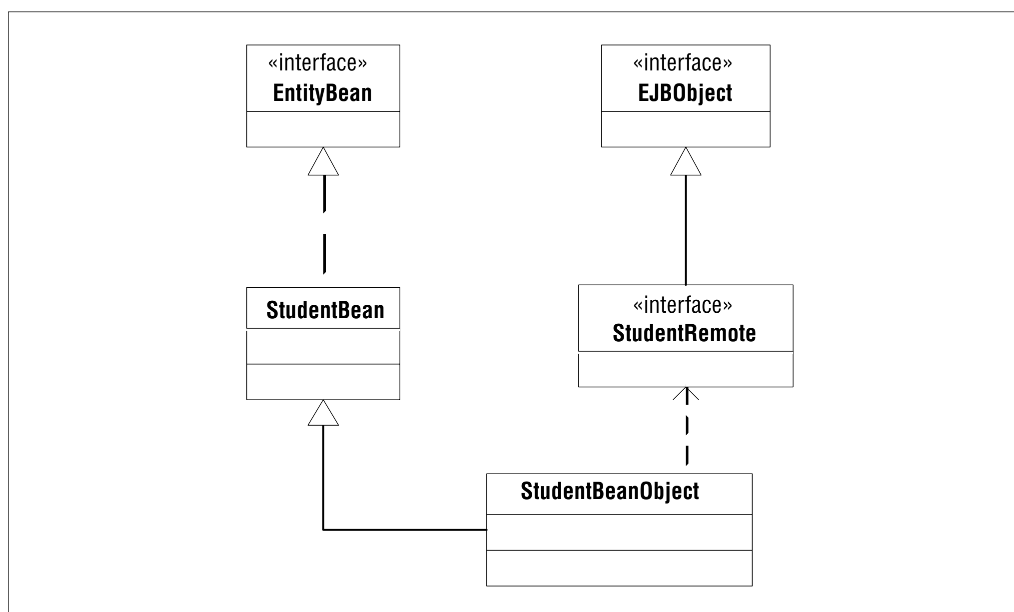


Figura 8.4 – *Architettura mista in cui lo StudentEJBObject implementa lo StudentBean e l'interfaccia Student. Questo caso può essere considerato come una variante del classico wrapper.*



Questa soluzione è rappresentata schematicamente in fig. 8.3. Il secondo schema invece prevede una soluzione mista, in cui l'oggetto remoto estende da una parte il bean, mentre dall'altra implementa l'interfaccia remota (fig. 8.4). Infine il terzo schema (fig. 8.5), si basa su una configurazione particolare, che di fatto non prevede la partecipazione del bean creato dallo sviluppatore. In questo caso è l'oggetto remoto che ha una implementazione proprietaria che rispecchia da un lato il server remoto, dall'altro i metodi del bean.

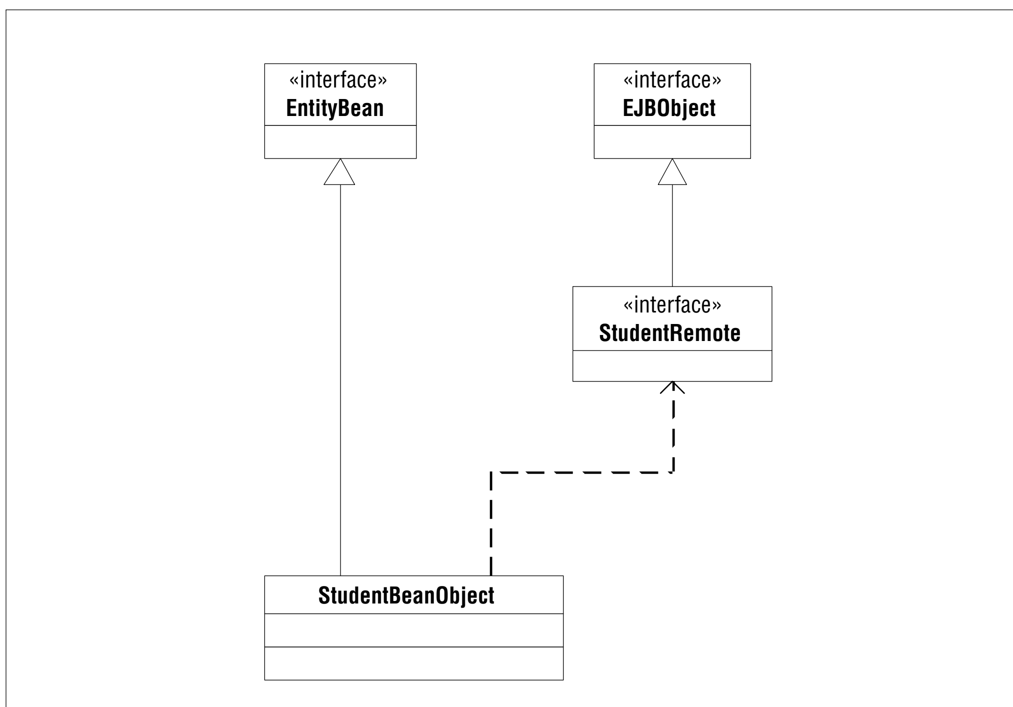
Anche la EJBHome viene generata automaticamente dal server, al momento dell'installazione del bean, quando vengono implementati tutti i metodi definiti nella home interface.

Ad esempio quando viene invocato il metodo create() sulla home interface, l'EJBHome crea una istanza dell'EJBObject, fornendogli una istanza di bean del tipo opportuno. Nel caso di entity bean, viene anche aggiunto un nuovo record nel database. Quando il metodo ejbCreate() ha terminato, l'EJBObject restituisce uno stub dell'oggetto remoto al client. Secondo il pattern Delegation, il client, utilizzando lo schema tipico di RMI, può utilizzare tale stub per invocare i metodi dell'EJBObject, il quale inoltrerà tali chiamate al bean stesso.

Utilizzare gli EJB

Si può passare a questo punto ad analizzare quali siano i passi necessari per l'utilizzo di un bean da parte di una applicazione client.

Figura 8.5 – *StudentEJBObject non implementa la classe StudentBean, ma la ingloba al suo interno secondo un modello proprietario del server.*



Per prima cosa è necessario ricavare un reference alla home interface, in modo da poter ricavare i vari stub degli oggetti remoti. Maggiori approfondimenti verranno dati in seguito, per il momento basti sapere che si utilizza per questo scopo la API Java Naming and Directory Interface (JNDI). Di seguito è riportato un pezzo di codice che su lato client permette di creare un oggetto student

```
public String createStudent(StudentDTO dto) {
  StudentRemote studentRemote = null;
  String studentId = null;

  try {
    Object student = context.lookup("StudentRemote");
    // ricava l'interfaccia home tramite JNDI
    StudentHome studentHome = (StudentHome) PortableRemoteObject. narrow(student, StudentHome.class);
    //  si crea lo studente con I parametri passati dal DTO
    studentRemote = studentHome.create(dto);
```

```
        // ricava l'id dello studente appena creato per poterlo passare al client invocante
        // è questa una operazione di business logic
        studentId = studentRemote.getId();
        return studentId;
      }
      catch (Exception e) {
        throw new EJBException(e);
      }
  }
```

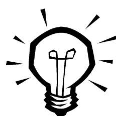
Si noti come dopo l'ottenimento della interfaccia home dello student si procede alla invocazione del metodo di creazione (in questa fase il server crea una riga nel database sottostante). L'uso del pattern DTO non è strettamente legato alla specifica EJB, ma fortemente consigliato nella realizzazione di una architettura complessa J2EE (si veda [EJBDesign]).

Da questo breve esempio si può intuire come in realtà l'utilizzazione di un bean si traduca nella sua istanziazione da parte del client e nel successivo utilizzo tramite invocazione dei metodi pubblici.

Lo scopo di un entity è quello di rappresentare un oggetto da utilizzare contemporaneamente da parte di più client: ogni entity bean è pensato per mantenere sincronizzati i dati fra i vari client, in modo da centralizzare le informazioni. Questo comporta una notevole riduzione della ridondanza delle risorse utilizzate dai vari client.

Gli entity bean rappresentano quindi un ottimo modo per modellare strutture dati secondo il paradigma a oggetti, ma non sono indicati per esprimere un'operazione o un determinato compito.

I session bean invece operano come agenti al servizio del client, effettuando tutte quelle operazioni che coinvolgono lo scambio di informazioni fra entity bean, e l'esecuzione di computazioni particolari, sia fini a se stesse, sia con effetto su altri entity bean, dando vita al workflow della applicazione.



Una regola non imposta dalla specifica ma dal buon senso e dalla buona progettazione del software dice che un entity non dovrebbe mai essere acceduto da un client remoto il quale invece dovrebbe sempre e solo comunicare con il session bean deputato alla gestione della applicazione.

In SchoolManager si trova per questo un session bean `SchoolManagerBean` che permette di accedere ai vari entity e di modificare lo stato della scuola (aggiungi studenti, insegnanti etc.). Per chi conosce la programmazione per pattern, lo `SchoolManagerBean` è di fatto il Session Façade della applicazione (vedi [EJBDesign]).

Qui di seguito è riportato come esempio il metodo `assignStudentToStudentClass` che mostra come assegnare uno studente ad una classe. Il metodo verrà invocato da un client il quale passerà al metodo id dello studente e id della classe.

```
public void assignStudentToStudentClass(String studentId, String studentCassId) {
  try {

    // si cerca la classe da assegnare allo studente
    Object studentClass = context.lookup("StudentClassRemote");
    StudentClassHome studentClassHome = (StudentClassHome)
                                        PortableRemoteObject. narrow(studentClass,
                                        StudentClassHome.class);
    // potrebbe dare errore se non si trova il record nel db
    StudentClass studentClass = studentClassHome.findByPrimaryKey(studentCassId);

    StudentRemote student = null;
    Object student = context.lookup("StudentRemote");
    StudentHome studentHome = (StudentHome) PortableRemoteObject.narrow(student, StudentHome.class);
    // potrebbe dare errore se non si trova il record nel db
    student= studentHome.findByPrimaryKey(studentId);
    student.setStudentClassId(studentClassId);
  }
  catch (Exception e) {
    throw new EJBException(e);
  }

}
```

In questo caso l'assegnazione studente-classe avviene tramite la semplice assegnazione dell'id della classe allo studente. In realtà questo modo di procedere non rispecchia il normale modo di gestire le relazioni (vedi oltre) ma è utile al momento per capire come i session e gli entity possano interagire fra loro.

L'habitat dei bean

L'ambiente all'interno del quale un bean vive e opera è composto da due oggetti, il container e il server: il primo è deputato al controllo del ciclo di vita del bean, il container rappresenta invece un concetto piuttosto vago, utilizzato esclusivamente per spiegare il funzionamento di EJB e per definirne il modello.

Definire un container equivale a specificare il comportamento e la modalità di interfacciamento (la cui base fondamentale sono le interfacce EntityBean e SessionBean) verso i vari oggetti remoti, specifica che deve essere adottata fedelmente dai vari costruttori di server EJB.

È libera invece l'implementazione della parte relativa al server, per permettere la massima libertà progettuale e implementativa ai vari costruttori di creare il loro prodotto EJB compliant.

Di fatto questo ha anche il grosso vantaggio di permettere la creazione di server EJB da distribuire come parte integrante di application server già presenti sul mercato ma non espres-

samente pensati in origine per supportare il modello EJB (vedi il caso di IBM WebSphere o BES di Borland).

I servizi di sistema

A partire dalla gestione delle varie istanze di bean, fino al controllo sulle transazioni, i servizi di sistema rappresentano probabilmente il punto più importante di tutta la tecnologia EJB, dato che permettono la cosiddetta separazione fra il lavoro del bean developer e quello del server.

Ad esempio senza la gestione dei pool di bean, si dovrebbe implementare manualmente tutta una serie di importanti aspetti, lavoro questo sicuramente lungo e insidioso, tale da rendere inutile l'utilizzo di uno strumento potente ma anche complesso come gli application server per EJB. Senza il supporto per i servizi di base EJB perderebbe tutta la sua importanza e non offrirebbe nessun motivo di interesse specie al cospetto di alternative come CORBA o RMI.

Di seguito si vedranno più o meno in profondità gli aspetti principali legati alla gestione dei servizi più importanti, cercando di dare una giustificazione del loro utilizzo e del loro funzionamento. In alcuni casi, come per esempio per le transazioni, verrà fatto un ulteriore approfondimento.

Resource management

Uno dei maggiori vantaggi offerto dal Component Transaction Model (CTM) è rappresentato dall'elevato livello di prestazioni raggiungibile in rapporto al carico di lavoro, ossia in funzione del numero di client che accedono ai vari oggetti remoti messi a disposizione dal server.

Per questo motivo il primo aspetto che si andrà a considerare è quello relativo alla gestione delle risorse condivise, ossia dei client che accedono ai bean installati sul server.

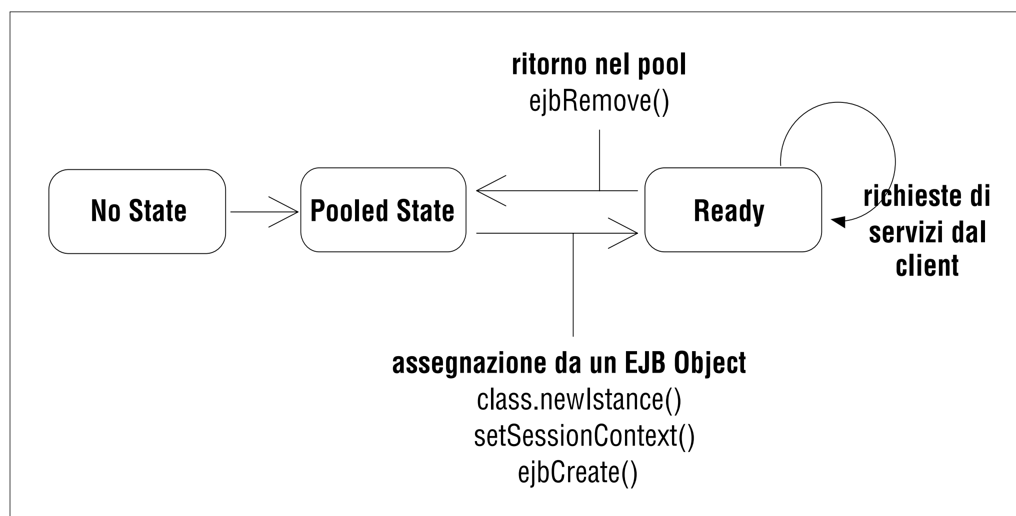
È bene tener presente che, come in tutti gli scenari concorrenti, specie nel caso di internet, il numero dei client in genere non è prevedibile. È quindi ovvio che tanto migliore sarà l'ottimizzazione della gestione e degli oggetti in esecuzione contemporanea ossia maggiore sarà la scalabilità della applicazione, tanto migliori saranno le prestazioni complessive del sistema.

Il meccanismo utilizzato in questo caso, preso in prestito dai sistemi di gestione di basi di dati, è quello del pooling delle risorse.

Alla base di tale organizzazione vi è la considerazione secondo la quale molto raramente si verifica l'ipotesi per cui tutti i client dovranno accedere nello stesso istante ai vari bean installati sul server: di rado si renderà quindi necessario istanziare e referenziare contemporaneamente tutti gli oggetti remoti che corrispondono ai molti client in funzione.

Oltre a una drastica riduzione del numero di connessioni aperte e degli oggetti attivi, il meccanismo del pool di oggetti remoti risulta essere anche più efficiente: infatti come si avrà modo di vedere più avanti, il meccanismo di condivisione risulta più efficiente rispetto alla creazione/distruzione sia di connessioni che di bean. In definitiva solo pochi oggetti remoti verranno realmente istanziati ed utilizzati, dando vita ad una politica di condivisione dei bean tramite le molte interfacce istanziate.

Figura 8.6 – *Ciclo di vita di un componente EJB nell'ambito del server.*



Questa separazione fra ciò che è in funzione sul lato server, e quello che invece il client gestisce è resa possibile dal fatto che il client non accede mai direttamente agli oggetti remoti, ma sempre tramite interfacce remote, secondo lo standard di EJB.

Pooling di entity bean

Per comprendere come sia messo in atto il sistema di pooling, si prenda prima in esame il caso degli entity bean. Tali componenti vivono il loro ciclo di vita passando dai seguenti stati:

- *No state*: è lo stato iniziale del bean, quando non è stato ancora istanziato e quando nessuna risorsa è stata ancora allocata per quel componente.

- *Pooled state*: il componente è stato istanziato dal container, ma ancora non ha nessun EJB Object associato.

- *Ready State*: infine il bean può essere ready, ossia è stato istanziato ed associato ad un EJB Object.

Sulla base di tale organizzazione risulta intuitivo come sia gestito il pooling: il server infatti istanzia una serie di componenti remoti, ponendoli nel pool, ed associando un componente remoto ad un determinato `EJBObject` solo al momento dell'effettivo bisogno.

Tutti i componenti remoti sono quindi equivalenti fra loro mentre sono nello stato pooled, ed acquistano una contestualizzazione solo al momento della invocazione da parte del client di

uno dei business methods del bean: il client effettua le invocazioni ai metodi dell'oggetto wrapper (EJBObject) e non direttamente dell'oggetto remoto.

Nella fase di *ready* il componente riceve le invocazione in callback dal server e non dal client.

Appena un componente viene istanziato e posizionato nel pool, riceve un nuovo context (istanza di un javax.ejb.EJBContext) che offre una interfaccia al bean per comunicare con l'EJB Environment.

L'EJBContext è in grado sia di reperire informazioni sul client che ha effettuato le invocazioni, sia di fornire un riferimento alle interfacce EJBHome ed EJBObject in modo da permettere l'invocazione da parte del bean stesso dei metodi di business di altri componenti.

Quando il bean remoto viene invalidato, nel caso in cui il client invochi uno dei metodi di rimozione, oppure perché il componente è uscito dallo scope di esecuzione, il bean viene separato dall'EJBObject e successivamente riposizionato nel pool.

Può anche accadere che un bean sia reimmesso nel pool dopo un periodo di inutilizzo prolungato da parte del client: in tal caso, nel momento in cui il client dovesse richiedere nuovamente l'utilizzo del bean, allora verrà effettuata nuovamente una operazione di assegnazione prelevando dal pool il primo componente disponibile.

Questa operazione detta *instance swapping* è particolarmente importante dato che permette di servire un alto numero di richieste utilizzando poche risorse attive: il tempo in cui il bean viene utilizzato dal client statisticamente è mediamente minore del tempo passato in attesa nel pool. Essendo l'instance swapping una operazione meno costosa della creazione di un bean, giustifica quindi l'utilizzo di un pool di bean.

Pooling di session bean

Nel caso degli stateless il meccanismo di pool è semplificato dalla natura stessa del componente: in questo caso infatti non viene mantenuta nessuna informazione fra due invocazioni dei metodi da parte del client, e quindi non è necessario memorizzarne lo stato. Ogni metodo esegue le istruzioni senza che si debba accedere in qualche modo ad informazioni memorizzate da qualche parte.

Questo permette al container di effettuare il processo di swapping senza tener conto di quale particolare istanza venga utilizzata.

Nel caso degli stateful session bean invece le cose sono leggermente differenti: l'integrità delle informazioni deve essere mantenuta in modo da ricostruire in ogni momento la storia delle varie invocazioni (che si definisce *conversational state*, ossia stato della conversazione fra client e bean); per gli stateful bean quindi non viene utilizzato il meccanismo di swapping di contesto.

In questo caso tutte le volte in cui è necessario utilizzare un componente, è sufficiente prelevarne uno vuoto dalla memoria (quindi in modo simile al sistema di pool), ripristinandone lo stato prelevando le informazioni direttamente da un sistema di memorizzazione secondaria (tipicamente file serializzati su file system). Questa operazione viene detta *attivazione*, ed è simmetrica a quella di scrittura su disco che prende il nome di *passivazione*.

Un aspetto piuttosto importante è quello legato agli eventuali campi definiti come transient: il meccanismo di salvataggio e ripristino dello stato si basa infatti sulla serializzazione, la quale,

come noto, impone il ripristino ai valori di default per i campi transient nel momento della deserializzazione dell'oggetto: per gli interi ad esempio il valore 0, per i reference null, per i boolean false e così via.

In questo caso però non si può avere la certezza dello stato dell'oggetto al suo ripristino, visto che la specifica EJB 1.0 lascia libertà di scelta all'implementatore del server. Per questo motivo, dato che l'utilizzo dei campi transient non è necessario tranne che in rare occasioni, se ne sconsiglia l'utilizzo.

Si ricorda che i metodi di callback utilizzati dal server per informare degli eventi di attivazione e passivazione sono i due ejbActivate() ed ejbPassivate(): il primo viene invocato immediatamente dopo l'attivazione, e il secondo immediatamente prima la passivazione del componente.

Entrambi sono particolarmente utili in tutti quei casi in cui si desideri realizzare una gestione particolareggiata delle risorse esterne al server; ad esempio nel caso in cui un bean debba utilizzare una connessione con un database o con un qualsiasi altro sistema di comunicazione.

Infine nel caso degli entity bean non si parla di mantenimento dello stato, ma piuttosto di persistenza delle informazioni memorizzate nel componente: i dati vengono in questo caso memorizzati in un database non appena qualche modifica viene effettuata nel componente.

La gestione della concorrenza

Quello della concorrenza è un problema molto importante, ma che per fortuna nel caso dei session bean non si pone, grazie alla natura stessa di questi componenti.

Ogni session stateful bean infatti è associato ad uno ed un solo client: non ha senso quindi pensare ad uno scenario concorrente, essendo il soggetto invocante sempre lo stesso.

Anche nel caso degli stateless session bean l'accesso concorrente perde di significato, dato che questo tipo di oggetti non memorizzano nessun tipo di stato, e quindi mai due bean potranno interferire fra loro.

Nel caso degli entity invece il problema della concorrenza si pone in tutta la sua drammaticità: questo tipo di bean infatti rappresenta un dato non legato ad un particolare client, dando luogo ad un tipico scenario di accesso concorrente.

La soluzione adottata in questo caso è la più conservativa, limitando l'accesso contemporaneo non solo a livello di risorsa specifica, ma anche, più precisamente, a livello di thread di esecuzione. Per thread di esecuzione si intende ad esempio una serie di soggetti in esecuzione conseguente (A invoca B che invoca C).

Fra le molte conseguenze di una scelta così conservativa, forse la più immediata è quella relativa alla proibizione dello statement synchronized, in modo da lasciare al server la gestione degli accessi concorrenti.

Questa politica, pur limitando notevolmente l'insorgere di possibili problemi, lascia scoperto il caso delle cosiddette chiamate rientranti o *loopback calls*. Si immagini ad esempio il caso in cui un client invochi un metodo di un bean A: in questo caso il thread di esecuzione facente capo al client acquisisce il lock associato al componente A.

Nel caso in cui il metodo del bean B debba a sua volta invocare un metodo di A, si otterrà un blocco di tipo deadlock, dato che B attende lo sblocco del lock su A per poter proseguire, evento che non si potrà mai verificare.

Figura 8.7 – *L'invocazione rientrante di un metodo può dar vita a problemi di deadlock.*

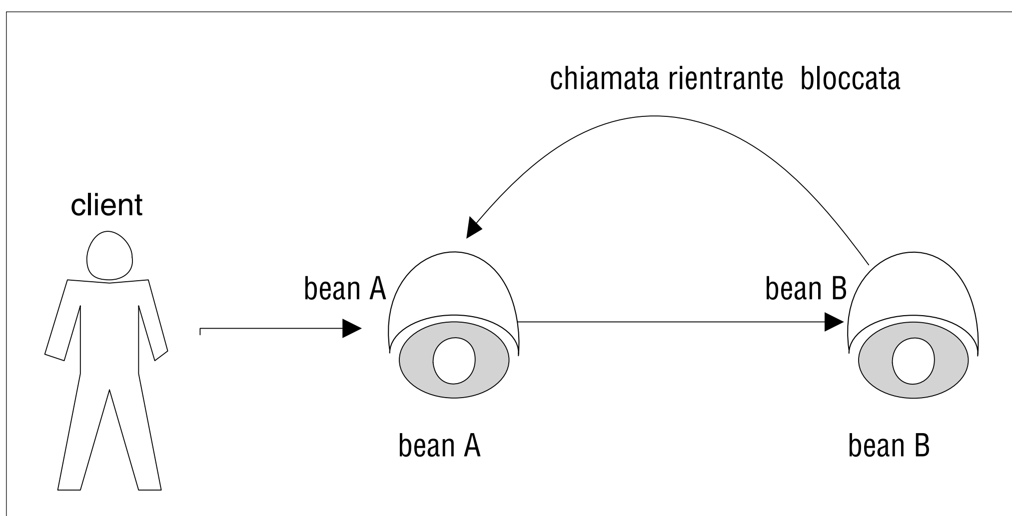
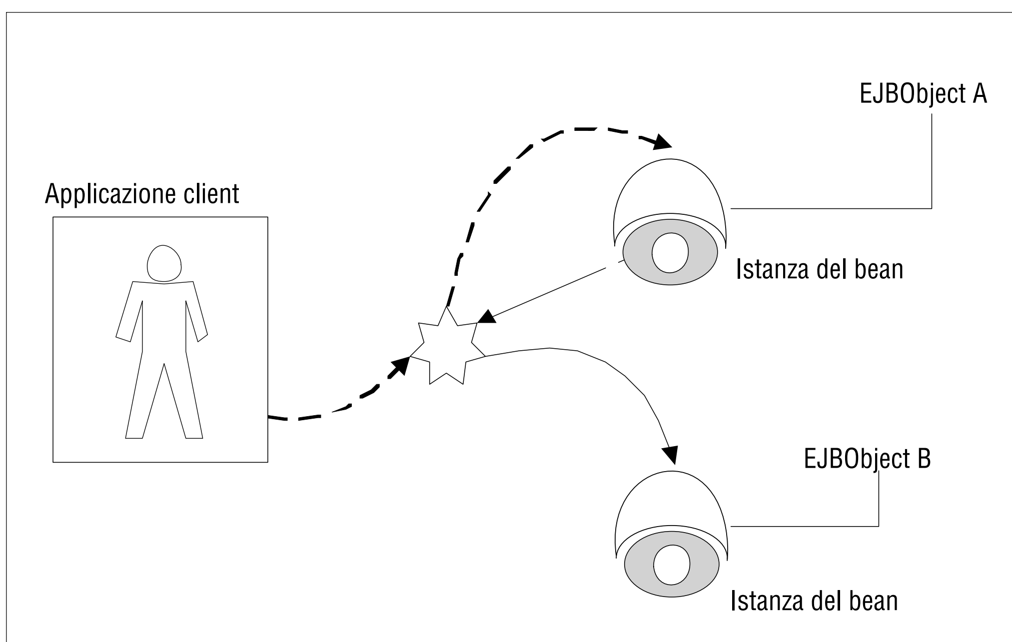


Figura 8.8 – *L'utilizzo di un componente, anche se locale, avviene sempre tramite il meccanismo dell'invocazione remota.*



Questo scenario nasce dalla scelta di voler attivare il lock a livello di thread di esecuzione, piuttosto che si singole entità. La soluzione a questo problema è quello di proibire il loopback: per i session la sua presenza porta ad una RemoteException, mentre gli entity possono essere configurati per funzionare anche in situazioni di loopback, anche se questa scelta viene fortemente scoraggiata. Ovviamente un componente può accedere ai suoi metodi (ossia un metodo di A può invocare senza problemi un altro metodo di A) senza dover sottostare ai vari vincoli di lock, come del resto avviene nel caso della classe Thread.

Ogni invocazione di un metodo di un bean avviene sempre e comunque secondo lo schema classico della invocazione remota basata su EJBObject. Quindi nell'esempio appena visto, il client C invoca i metodi di A secondo lo schema visto: conseguentemente A diviene client di B, e quindi invoca i suoi metodi ricavandone l'interfaccia remota. B a sua volta nella chiamata di loopback ritorna ad essere client di A e quindi, grazie al meccanismo di lock sui thread, non può eseguire tale operazione. Sebbene questa organizzazione possa apparire eccessivamente complessa, presenta due importanti vantaggi: da un lato permette di mantenere un elevato livello di standardizzazione, dato che ogni componente deve essere progettato ed implementato seguendo una rigida specifica standard. Dall'altro, visto che A invoca i metodi di B tramite una interfaccia remota, è possibile effettuare sostituzioni e migrazioni del bean da un server ad un altro, senza che il client ne abbia percezione. Questo permette massima flessibilità, semplicità di gestione e con poco lavoro consente l'implementazione di tecniche di load balancing fra server differenti. Passare per forza sempre dall'EJBObject rappresenta però anche il principale tallone d'Achille di tutta l'architettura EJB: ogni invocazione infatti è vista come una invocazione remota alla RMI anche se l'oggetto è in esecuzione localmente (come potrebbe essere nel caso di B) e questo può provocare grossi rallentamenti in fase di esecuzione.

Per questo motivo fino al rilascio della versione EJB 2.0 tutti gli application server avevano introdotto tecniche proprietarie nascoste e trasparenti di "invocazione ottimizzata" dei metodi remoti, passando da invocazioni remote-RMI (che passano per lo strato di rete) a invocazioni locali in-JVM. Sun, prendendo atto di questa situazione ha introdotto con EJB 2.0 il concetto di interfaccia locale, di cui si parlerà più avanti.

Le transazioni

Il modello di EJB fornisce un robusto supporto per le transazioni. Anche in questo caso il programmatore del componente non deve implementare nessuna struttura particolare, dato che è sufficiente definire i dettagli operativi in fase di deploy del componente: indicando tramite file XML quali operazioni dovranno essere atomiche e specificando un contenitore di transazioni, si può fornire il supporto transazionale per un intero bean piuttosto che per i singoli metodi. Vista l'importanza e la complessità dell'argomento, si rimanda l'approfondimento di questi aspetti al paragrafo relativo presente più avanti nel capitolo.

La gestione della persistenza

La gestione della persistenza delle informazioni è un aspetto molto importante, anche se riguarda esclusivamente il caso degli entity bean. Questi oggetti, di fatto, rappresentano dati o

collezione di dati che possono essere memorizzati tramite l'utilizzo di un database di vario tipo. Tale sincronizzazione con la memoria secondaria permette fra le altre cose, di realizzare applicazioni robuste e fault tolerant: ad esempio in seguito ad un blocco del sistema, ogni componente potrà riassumere la sua originaria configurazione rileggendo i dati memorizzati nel database.

Come si è precedentemente accennato la persistenza dei dati può essere effettuata dal container o direttamente dal bean.

Come presumibilmente appare ovvio la prima soluzione è quella più comoda e sicura, mentre la bean managed viene utilizzata ad esempio quando sia necessaria una personalizzazione del processo di salvataggio dei dati, oppure quando si debba comunicare con codice legacy.

In questo caso una soluzione basata su EJB risulta essere particolarmente importante ed efficace dato che permette di esportare una interfaccia Java di alto livello su un sistema legacy basato su tecnologie obsolete o comunque non standard. I vari componenti remoti funzioneranno come contenitori in comunicazione con la parte legacy ad esempio via socket o per mezzo di JNI.

Mapping dei dati e modello di memorizzazione

Il legame fra bean e dati memorizzati nel database è talmente stretto che una modifica nel primo si riflette in un aggiornamento dei dati: viceversa inserire a mano un nuovo record (ad esempio tramite uno statement SQL nel caso in cui si sia optato per un database relazionale) corrisponde a creare una istanza logica di un nuovo bean, anche se non necessariamente tale operazione produce realmente nuove classi Java.

Il database da utilizzare per la memorizzazione di un entity può essere di tipo relazionale ad oggetti o di altro tipo.

Il modello relazionale è quello adottato nella maggior parte dei casi a causa delle maggiori prestazioni, affidabilità e maturità dei prodotti rispetto ad altre tecnologie, e soprattutto perché è più conosciuto dagli utilizzatori finali.

La grossa limitazione di questo modello è quello di essere estraneo alla logica ad oggetti utilizzata in EJB ed in Java in genere: si rende necessario quindi uno strato di software con cui effettuare il mapping relazionale-object oriented, in modo da trasformare i campi di un bean in valori delle colonne di un database.

Purtroppo non sempre è possibile effettuare in modo lineare ed automatico questo passaggio, tanto che in molti casi si rendono necessarie conversioni manuali, come ad esempio nel caso in cui particolari oggetti Java non siano direttamente convertibili nel formato accettato dal database. Inoltre la struttura relazionale può complicarsi molto in funzione alla reale struttura del bean di partenza.

I database ad oggetti invece non richiedono nessuna operazione di conversione, e quindi permettono ad un componente di essere salvato in modo diretto nel database stesso, consentendo di limitare la complessità della struttura.

Lo svantaggio in questo caso risiede nella immaturità dei vari database ad oggetti, nelle prestazioni non paragonabili con il modello relazionale e nella mancanza di una diffusa cultura di tali strumenti fra gli sviluppatori e utilizzatori.

Servizio di naming

Qualsiasi sistema di programmazione distribuita ha alla base un qualche meccanismo che permetta di rintracciare oggetti remoti in esecuzione in spazi di indirizzamento eterogenei. Un sistema di questo tipo si basa sui cosiddetti *naming services* che sono essenzialmente di due tipi: servizio di bind (ossia registrazione di un oggetto remoto in un qualche registry tramite nome logico) e servizio di lookup (che corrisponde alla operazione inversa, ossia ricercare la particolare istanza di un oggetto remoto, partendo da un nome logico).

Nel caso di EJB il meccanismo utilizzato è quello offerto dalla Java Naming and Directory Interface (JNDI API). Questo strumento implementa una astrazione gerarchica ad alto livello di una ipotetica collezione di risorse (file, oggetti, stampanti, devices varie). Per una trattazione approfondita del funzionamento di JNDI si può far riferimento alla bibliografia (vedi [JNDI]) o al capitolo dedicato all'argomento.

JNDI non rappresenta l'unico strumento di naming utilizzabile all'interno di un server EJB, anche se la specifica dice che ogni produttore di server deve fornire almeno una interfaccia di questo tipo ai vari client che intendano utilizzare i bean installati.

Da questo punto di vista il client deve utilizzare la JNDI API per iniziare la connessione verso un EJB Server specificando l'interfaccia EJBHome dell'oggetto remoto che si intende utilizzare. Ad esempio si potrebbe scrivere

```
import javax.naming.*;
...
Context jndiContext = new  InitialContext(props);
MyHome home = (MyHome) jndiContext.lookup(beanName);
MyBean  myBean =Home.create(param1, param2);
myBean.doSomething();
```

In questo caso props, istanza di Properties, indica al servizio JNDI dove si trova il server EJB da utilizzare; beanName invece indica il nome logico con cui il componente è stato registrato.

In genere ogni application server fornisce nel pacchetto di documentazione tutti i parametri di connessione e di inizializzazione del contesto. Per maggiori informazioni si faccia riferimento alla documentazione del prodotto utilizzato.

Security

La gestione della sicurezza è uno degli aspetti chiave su cui si basano la maggior parte dei moderni sistemi di comunicazione e di programmazione distribuita. Si può intendere la sicurezza a tre livelli: autenticazione, access control e sicurezza nella comunicazione.

Nel primo caso, grazie ad un qualche sistema di riconoscimento si deve consentire a un utente (persona fisica) o entità (ad esempio un oggetto) l'accesso al sistema e la possibilità di usufruire dei servizi messi a disposizione. Questo sistema in genere è considerato non sufficientemente flessibile, dato che permette di attivare o disattivare del tutto il set delle possibili operazioni effettuabili. Per questo spesso si parla di meccanismo On/Off.

Il riconoscimento può avvenire tramite password, certificato digitale o tramite meccanismi particolari come smart card o device elettromagnetiche.

Anche se in EJB non è formalmente specificato nessun meccanismo per permettere l'autenticazione, grazie ai servizi offerti da JNDI, si può comunque implementare un proprio sistema di riconoscimento: ad esempio si potrebbero utilizzare le informazioni per il login al momento in cui si effettua l'operazione di lookup semplicemente passando i dati utente come proprietà al context che effettua la ricerca; di seguito ecco un breve esempio

```
import javax.naming.*;
...
Properties props = new Properties();
props.put(Context.SECURITY_PRINCIPAL, uid);
props.put(Context.SECURITY_CREDENTIALS, passwd);
Context jndiContext = new  InitialContext(props);
...
```

Nel caso si possa disporre di un sistema di Access Control, allora sarà possibile definire una policy di permessi in modo da autorizzare un utente ad effettuare determinate operazioni. Questa tipologia di security è quella inclusa nella specifica 1.0 di EJB.

In questo caso ad ogni client viene associato un elemento di istanza di `java.security.Identity` che rappresenta l'identità con cui il client potrà effettuare le operazioni con gli oggetti remoti. Una Identity rappresenta un utente o un ruolo nel caso in cui l'oggetto remoto sia invocato da un altro bean, che come specificato in precedenza assume il ruolo di client. Questa puntualizzazione è molto importante, dato che colma una carenza presente molto spesso in sistemi distribuiti: ad esempio ci si potrebbe chiedere nel caso di una servlet, con quali diritti esso sia mandato in esecuzione (generalmente con quelli dell'utente che ha fatto partire il servlet container, ma non è detto).

La `Identity` associata ad un client viene utilizzata in modo del tutto trasparente: ad esempio quando il client richiede l'utilizzo di un particolare metodo, alla invocazione dello stesso viene passata anche l'istanza della identità associata, al fine di effettuare il controllo sui diritti. La definizione delle varie policy si effettua al momento del deploy grazie al `DeploymentDescriptor` il quale contiene le varie istanze di `ControlDescriptor` ed `AccessControlEntry`: questi ultimi permettono di definire la lista degli utenti ammessi ad effettuare determinate azioni, mentre i `ControlDescriptor` consentono di specificare il cosiddetto `runAs` di ogni `Identity`, ossia con quale identità ogni metodo potrà essere eseguito.

Ad esempio si può specificare che un metodo qualsiasi di un determinato componente possa essere eseguito solo dall'utente "pippo", ma che tale metodo poi possa essere eseguito nel sistema come utente "amministratore di sistema". Si intuisce quindi come tutti questi meccanismi possano garantire un altissimo livello di personalizzazione del sistema in fatto di sicurezza. Il prezzo da pagare è ovviamente la complessità del tutto, come già in passato è avvenuto con l'introduzione dei sistemi avanzati di crittografia in Java, o con l'avvento del processo di firma digitale delle applet.

Infine si può intendere la sicurezza come quel meccanismo atto a rendere le comunicazioni non decifrabili da estranei (in EJB l'invocazione dei metodi ed i parametri passati). Quasi sem-

pre questo si traduce nel dover utilizzare sistemi di crittografia basati su protocolli particolari. Al momento nella specifica EJB non è stata formalmente definita nessuna tecnica per supportare meccanismi di questo tipo.

Gli entity bean

Lo scopo principale di un entity bean è quello di memorizzare delle informazioni ed offrire al contempo una serie di metodi per la gestione da remoto di tali dati. In genere gli entity bean dovrebbero limitare al massimo le funzionalità di business logic, offrendo solamente metodi di accesso e modifica dei dati che essi stessi rappresentano. Uno degli aspetti più importanti degli entity bean è come questi implementano la persistenza dei dati all'interno del database. Due sono le possibilità: da un lato tutto il processo di sincronizzazione viene gestito dal container in modo trasparente, mentre nell'altro caso tutte le informazioni sono salvate su disco direttamente tramite operazioni effettuate dal bean. Nel primo caso si parla di bean Container Managed Persistence (CMP), mentre nell'altro si utilizza la definizione Bean Managed Persistence (BMP). Per il momento verrà effettuata una breve introduzione a tali concetti, rimandando al paragrafo dedicato alla parte pratica per una analisi più dettagliata.

In un bean di tipo CMP il server si preoccupa in modo del tutto trasparente di sincronizzare le informazioni contenute nel bean al variare dello stato e soprattutto in funzione dello stato assunto all'interno del ciclo di vita del componente. Tranne per il caso delle variabili definite transient, tutti gli oggetti serializzabili, o le variabili primitive possono essere salvate. Dato che, nella maggior parte dei casi il database utilizzato è di tipo relazionale, un utilizzo di variabili primitive sicuramente semplifica questo passaggio.

Gli entity bean di tipo CMP possono risultare particolarmente utili nel caso in cui sia semplice effettuare un mapping bean-database: quando il processo di persistenza del componente richiede invece un trattamento particolare, sia per la maggiore complessità del caso, sia per la necessità di utilizzare tecniche di persistenza particolari, si può passare ad implementare una gestione manuale della sincronizzazione fra componenti e dati nel database. Il prezzo da pagare in questo caso è dato dalla maggiore complessità del codice del bean, anche se il passaggio da CMP ad un BMP è relativamente semplice grazie alla filosofia di base del modello EJB: il programmatore infatti dovrà semplicemente implementare i vari metodi di callback invocati dal server (come `ejbCreate()` o come quelli di sincronizzazione `ejbStore()` ed `ejbLoad()` e quelli di ricerca) al fine di definire "cosa" deve essere fatto in corrispondenza di un determinato evento; il "quando" questo deve essere fatto è demandato al server che gestisce il bean ed il suo ciclo di vita. Di seguito sono riportate le considerazioni più importanti relative ad ognuno di questi metodi.

Metodi di creazione

Il metodo

```
ejbCreate()
```

viene invocato indirettamente dal client quando questo invoca il metodo `create()` della interfaccia home, ossia quando genera un nuovo componente. Nel caso dei BMP questo metodo è respon-

sabile di creare i dati relativi al bean all'interno del database. Il metodo create ritorna sempre la chiave del bean appena creato.

Da un punto di vista strettamente implementativo, nel caso in cui il database d'appoggio scelto sia di tipo relazionale, all'interno del metodo troveremo delle istruzioni insert SQL, che effettueranno una o più insert delle varie variabili d'istanza del componente. Si ricordi che tutte le eccezioni SQL dovranno essere inglobate in wrapper e propagate come `EJBExceptions` o `RemoteExceptions`. Il metodo `ejbPostCreate()` invece, non è coinvolto dal processo di persistenza utilizzata e deve sempre ritornare `void`.

Metodi di sincronizzazione

I metodi

```
ejbLoad()
ejbStore()
```

sono invocati tutte le volte che il container decide che sia necessaria una sincronizzazione fra i dati memorizzati nel database e le variabili del componente. Il primo verrà invocato in concomitanza di una transazione o prima di un business method, in modo da avere la certezza che il bean contenga i dati più recenti contenuti nel database.

Il corpo del metodo non sarà molto differente da `ejbCreate()`, tranne che in questo caso, sempre nell'ipotesi di utilizzare un database relazionale, si troveranno delle update SQL piuttosto che delle insert. Questi metodi non sono esposti nella interfaccia di home dato che essi sono invocati dal container in concomitanza di particolari eventi non gestibili direttamente dal client. Continuano a valere anche in questo caso le considerazioni sulla gestione in wrapper delle eccezioni.

Il metodo

```
ejbRemove()
```

corrisponde al `remove` della home interface e provoca la rimozione del bean dal container (e la conseguente cancellazione dei dati dal database).

Metodi di ricerca

Per ogni metodo di ricerca presente nella home interface, dovrà esserne presente uno analogo nel bean. Si ricordi infatti che il client effettua le invocazioni direttamente sulla home che poi inoltra tali chiamate direttamente al bean. Nel caso dei BMP, i metodi di ricerca sono responsabili di trovare nel database i vari record che corrispondono ai criteri di ricerca impostati.

Ad esempio se nella home è presente codice del tipo

```
public interface MyHome extends javax.ejb.EJBHome{
  public MyBean findByPrimaryKey(MyBeanPK pk) throws FinderException, RemoteException;
  public Enumeration findByName(String name)  throws FinderException,  RemoteException;
}
```

allora nel bean si dovrà scrivere

```
public class MyBean extends javax.ejb.EntityBean{
   public MyBeanPK ejbFindByPrimaryKey(MyBeanPK pk) throws FinderException, RemoteException{}
   public Enumeration ejbFindByName(String name) throws FinderException, RemoteException{}
}
```

La FinderException è stata definita appositamente per segnalare l'insorgere di problemi durante la fase di ricerca.

Questi metodi appena visti sono quelli tramite i quali il server avverte il bean di un passaggio di stato o di un cambiamento all'interno del ciclo di vita. La loro definizione è obbligatoria per correttezza, ma non è necessario implementarne il comportamento.

Se il bean in esame è un CMP allora non vi è la necessità di implementare il corpo dei metodi ma solo di specificarne le regole di comportamento tramite XML. A prima vista l'implementazione o meno di tali metodi è quindi la principale differenza fra un CMP ed un BMP. Dal punto di vista del client invece e di tutto il resto dell'applicazione, non vi è nessuna differenza fra utilizzare uno o l'altro tipo di bean.

La specifica EJB 1.1 permette di rendere persistenti anche i campi di un bean che contengano riferimenti ad altri bean: in questo caso il costruttore del server deve implementare alcune tecniche piuttosto complesse al fine di effettuare il mapping più adatto possibile. Normalmente questo si traduce nella memorizzazione della chiave primaria (oggetto PrimaryKey), degli oggetti Handle o HomeHandle, o di altri riferimenti che identifichino univocamente il bean contenuto.

La gestione della persistenza a carico del container semplifica molto il lavoro di implementazione del bean, ma complica non poco la fase di progettazione del server; infatti in questo caso le tecniche di persistenza devono essere il più generiche e automatiche possibili, visto che non si può fare nessuna assunzione a priori su come sia fatto il componente.

Nel caso invece dei bean managed si possono implementare tecniche ad hoc a seconda del caso. Come si potrà vedere in seguito, tale soluzione è concettualmente molto semplice, dato che si riconduce alla scrittura di porzioni di codice JDBC all'interno dei vari metodi invocati in modalità callback durante il ciclo di vita del bean.

Identificazione di un bean: la primary key

Quando un client effettua una operazione di lookup su un oggetto remoto, si deve poterlo identificare in maniera univoca fra tutti quelli messi a disposizione dal server. Per questo motivo ad ogni bean è associata una chiave, la PrimaryKey, che può essere costituita da una classe apposita, oppure da un campo del bean stesso. Nel primo caso, la classe generata ha un nome che segue lo schema NomeBeanPK, e contiene al suo interno tutte le informazioni necessarie per individuare il bean. Ad esempio riconsiderando il bean Student, si potrebbe scrivere:

```
public void StudentPK{
   public int studentId;
   public StudentPK (){}

   public StudentPK (int id){
```

```
      studentId = id;
   }

   public boolean equals(Object obj){
      if (obj == null || !(obj instanceof StudentPK))
         return false;
      else {
         if(((StudentPK)obj). studentId == this. studentId)
            return true;
         else
            return false;
      }
   }

   public int hashCode(){
      return this. studentId
   }

   // converte  in stringa il valore della chiave
   public String toString(){
      return this. studentId + "";
   }
}
```

In questo caso questa classe funge da wrapper per una variabile intera che rappresenta la chiave del bean. Per questo motivo sono stati ridefiniti i metodi equals() ed hashCode() in modo da operare su tale variabile: nel primo caso viene fatto un controllo con la chiave dell'oggetto passato, mentre nel secondo si forza la restituzione di tale intero (cosa piuttosto comoda ad esempio nel caso in cui si vogliano memorizzare i bean in una tabella hash, per cui due codici hash potrebbero essere differenti anche se fanno riferimento allo stesso id).

Si noti la presenza del costruttore vuoto, necessario per poter permettere al container di istanziare un nuovo componente in modo automatico e di popolarne gli attributi con valori prelevati direttamente dal database utilizzato per la persistenza. Questa operazione effettuata in automatico, viene eseguita piuttosto frequentemente durante il ciclo di vita del componente, ed è alla base della gestione della persistenza ad opera del container.

Per questo motivo tutti i campi della PrimaryKey devono essere pubblici, in modo che tramite la reflection il container possa accedervi in modo automatico. Alcuni server permettono di accedere, tramite tecniche alternative, anche a variabili con differente livello di accesso, ma questo diminuisce la portabilità della chiave.

Inoltre una PrimaryKey deve sottostare alle regole imposte per l'invocazione remota basata su RMI: questo significa che possono essere chiavi quelle classi serializzabili o remote in cui siano stati ridefiniti i metodi equals() ed hashCode().

In alternativa all'utilizzo di PrimaryKey è possibile utilizzare direttamente dentro il bean campi di tipo String o wrapper di tipi primitivi (ad esempio Integer): in questo caso si parla di chiave semplice. Il metodo di ricerca in questo caso agisce direttamente sul campo del bean.

In tale situazione la chiave primaria non può essere un tipo primitivo, ma occorre utilizzare sempre un wrapper, al fine di garantire una maggiore standardizzazione della interfaccia: ad esempio il metodo getPrimaryKey() restituisce una istanza di Object, che verrà convertita a seconda del caso. L'utilizzo di una classe apposita come chiave è da preferirsi se si desidera consentire ricerche basate su chiavi composte (uno o più campi), oppure quando sia necessaria una maggiore flessibilità. La specifica EJB 1.0 lasciava indefinito questo aspetto, rimandando al costruttore del server la scelta di supportare o meno le chiavi semplici.

Con la versione 1.1 invece la scelta di quale campo utilizzare non viene fatta durante la progettazione del componente, ma piuttosto durante la fase di deploy grazie ad un apposito documento XML. Ad esempio si potrebbe decidere di scegliere un campo studentId contenuto nel bean stesso tramite la seguente sequenza di script XML:

```
<enterprise-beans>
  <entity>
    <primary-field>studentId</primary-field>
  </entity>
</enterprise-beans>


  table= new MyTable(strTable);
}


  ...
}
```

Il poter definire in fase di deploy quale sia la chiave da utilizzare ha una importante impatto sulla portabilità: in EJB 1.0 infatti il progettista doveva decidere a priori quale sarebbe stato il campo da utilizzare per tale scopo, dovendo quindi fare una serie di assunzioni sul database utilizzato per la persistenza (relazionale o ad oggetti) e sulla struttura dati da esso offerta (ad esempio struttura delle tabelle).

Rimandare alla fase di deploy tale scelta di fatto permette di svincolare completamente la fase di progettazione ed utilizzazione del componente dalla quella di utilizzo finale. È per questo motivo quindi che tutti i metodi che hanno a che fare con la chiave lavorano con parametri di tipo Object, rimanendo così slegati dalla particolare scelta sul campo scelto come chiave.

I metodi di ricerca

La ricerca di un bean, una volta definite le chiavi, avviene tramite i metodi di ricerca messi a disposizione dalla interfaccia remota. La specifica infatti dice che tale interfaccia deve fornire

zero o più metodi creazionali, ed uno o più metodi di ricerca. Nel caso dei CMP, i metodi di ricerca sono implementati automaticamente al momento del deploy, in base alla politica particolare implementata dal container per effettuare la ricerca, ed in base a quanto specificato nei parametri di deploy. In alcuni casi il server permette di specificare attraverso il tool di deploy anche il comportamento di tali metodi.

Il findByPrimaryKey() è il metodo standard, ossia quello che consente la ricerca in base alla chiave primaria, ma non è detto che sia il solo. Ad esempio potrebbe essere comodo effettuare ricerche sulla base di particolari proprietà del bean: in questo caso i nomi dei metodi seguono lo schema findByNomeCampo(). Nel caso di ricerche per campi differenti dalla chiave primaria si possono avere risultati multipli: per questo con la specifica 1.1 il risultato della ricerca può essere un oggetto di tipo Collection. I metodi di ricerca che ritornano un solo oggetto devono generare una eccezione di tipo FinderException in caso di errore, mentre una ObjectNotFoundException nel caso in cui non sia stato trovato niente.

La sincronizzazione con il database: metodi ejbLoad() ed ejbStore

I metodi ejbLoad() ed ejbStore() possono essere utilizzati per effettuare tutte quelle operazioni di manipolazione dei dati relative al momento di persistenza del bean. Nei BMP questo significa che conterranno le istruzioni SQL per la scrittura e lettura dei dati. Anche nel caso di un CMP tali metodi possono essere d'aiuto, ad esempio nel caso in cui vi sia la necessità di effettuare conversioni o preparazioni del bean prima che questo sia reso persistente, oppure subito dopo la lettura dal database.

Si pensi al caso in cui uno dei campi del bean sia costituito da un oggetto non serializzabile: nel pezzo di codice che segue si è preso in esame il caso di una ipotetica classe MyTable non serializzabile.

```
public class MyBean extends EntityBean {

  public transient MyTable table;
  public String strTable;

  public void ejbLoad(){
    strTable=table.toString();
  }

  public void ejbStore(){
    table= new MyTable(strTable);
  }

  ...
}
```

In questo caso la variabile MyTable è indicata transient, in modo da impedirne la memorizzazione automatica nel database e potere salvare invece la rappresentazione testuale della tabella tramite l'oggetto strTable.

Senza entrare nei dettagli dell'implementazione della classe MyTable, sarà sufficiente dire che il costruttore accetterà come parametro una stringa contenente i valori riga-colonna che costituiscono la tabella, e che il metodo toString() effettuerà l'operazione opposta.

Anche se l'esempio è volutamente poco significativo da un punto di vista applicativo, mostra con quale semplicità sia possibile implementare uno strato di pre- e postelaborazione relativamente all'operazione di lettura e scrittura su database.

L'interfaccia EntityBean

L'interfaccia EntityBean definisce una serie di metodi di callback atti alla gestione dello stato di un bean.

```
public interface EntityBean extends EnterpriseBean{
  public abstract void ejbActivate() throws RemoteExcetpion;
  public abstract void ejbPassivate() throws RemoteExcetpion;
  public abstract void ejbLoad() throws RemoteExcetpion;
  public abstract void ejbStore() throws RemoteExcetpion;
  public abstract void ejbRemove() throws RemoteExcetpion;
  public abstract void setEntityContext(EntityContext ec) throws RemoteExcetpion;
}
```

La maggior parte di tali metodi però non sono particolarmente utili nel caso in cui sia il container a dover gestire la persistenza dei dati memorizzati nel componente, fatta eccezione per ciò che riguarda la gestione del contesto.

L'interfaccia EntityContext fornisce tutte le informazioni utili durante il ciclo di vita di un bean, indipendentemente dalla politica adottata per il mantenimento dello stato, e rappresenta il tramite fra il bean ed il proprio container.

Il primo metodo invocato successivamente alla creazione della istanza del bean è il setEntityContext() il quale passa al componente una istanza del contesto in cui agisce: esso viene invocato prima che il bean appena creato entri nel pool delle istanze pronte per l'utilizzo.

Ripensando al ciclo di vita di un bean, visto in precedenza, si potrà rammentare come in questa situazione le varie istanze non sono ancora state popolate con i dati prelevati dal database, rendendole di fatto tutte equivalenti. È al momento della invocazione da parte del client che un bean viene prelevato dal pool e contestualizzato: le sue proprietà vengono istanziate, con dati prelevati dal database, e l'istanza viene associata o "wrapperizzata" con un EJBObject per l'interfacciamento con il client. In questa fase viene creato un contesto per il bean tramite la creazione e l'assegnazione di un oggetto EntityContext, che rappresenta il tramite fra il bean e l'EJBObject.

Il contesto inoltre è particolarmente importante se si riconsidera il meccanismo di pool e di swap di istanze dei vari bean: da un punto di vista implementativo infatti lo swapping è reso

possibile grazie al continuo cambio di contesto associato al bean stesso il quale non necessita di sapere cosa stia succedendo fuori dal suo container o quale sia il suo stato corrente. Il bean in un certo senso "vive" il mondo esterno grazie al filtro del proprio contesto.

Fino a quando il metodo ejbCreate() non ha terminato la sua esecuzione, nessun contesto è assegnato al bean e quindi il bean non può accedere alle informazioni relative al suo contesto come la chiave primaria o dati relativi al client invocante: tali informazioni sono invece disponibili durante l'invocazione del metodo ejbPostCreate().

Invece tutte le informazioni relative al contesto di esecuzione come ad esempio le variabili d'ambiente sono già disponibili durante la creazione.

Quando un bean termina il suo ciclo di vita, il contesto viene rilasciato grazie al metodo unsetEntityContext().

Per quanto riguarda invece i metodi creazionali, benché acquistino particolare importanza nel caso di gestione della persistenza bean managed, possono rivelarsi utili per ottimizzare la gestione del componente e delle sue variabili.

Ogni invocazione di un metodo di questo tipo sulla home interface viene propagata sul bean in modo del tutto analogo a quanto avviene per i metodi di business invocati sulla remote interface. Questo significa che si dovrà implementare un metodo ejbCreate() per ogni metodo corrispondente nella home interface.

La loro implementazione così come la loro firma dipende da quello che si desidera venga compiuto in fase di creazione. Tipicamente si tratta di una inizializzazione delle variabili di istanza.

Ad esempio si potrebbe avere:

```
// versione EJB 1.1 che ritorna un bean
public StudentBean ejbCreate(int id, String fName,  String lName, ...){
   this.studentId=id;
   this. firstName = fName;
   this. lastName = lName;
   ...
   return null;
}

// versione EJB 1.0  che ritorna un void
public void ejbCreate(int id, String name, ...){
   this.beanId=id;
   this.beanName=name;
   ...
}
```

Come si può notare la differenza fondamentale fra la specifica EJB 1.0 e 1.1 è il valore ritornato. Nella 1.0 infatti il metodo crea un componente senza ritornare niente, mentre nella 1.1 viene ritornato una istanza null della classe che si sta creando. Sebbene il risultato finale sia simile, dato che il valore ritornato viene ignorato in entrambi i casi, questa soluzione ha una

importante motivazione di tipo progettuale: la scelta fatta nella specifica 1.1 permette infatti un più semplice subclassing del codice consentendo l'estensione di CMP da parte di un BMP. Nella versione precedente questo non era possibile dato che in Java non è permesso l'overload di un metodo che differisca solo per il tipo ritornato.

La specifica 1.1 permette quindi ai costruttori di server di supportare la persistenza container managed semplicemente estendendo un bean container managed con un bean generato di tipo bean managed.

Con la CMP 2.0 questa sottile differenza ha perso di importanza dato il grosso livello di complessità che caratterizza il nuovo modello di persistenza.

La gestione del contesto

Il contesto in cui vive un bean, comune a tutti i tipi di componenti, è dato dalla interfaccia EJBContext dalla quale discendono direttamente sia la EntityContext che il SessionContext.

Ecco la definizione di tale interfaccia secondo la specifica 1.1:

```
public interface EJBContext{

  public ejbHome getEJBHome();

  // metodi di security
  public java.security.Principal getCallerPrincipal();
  public boolean isCallerInRole(String roleName);

  // metodi deprecati
  public java.security.Identity getCallerIdentity();
  public boolean isCallerInRole(Identity role);
  public Properties getEnvironment();

  // metodi transazionali
  public UserTransaction getUserTransaction(); throws IllegalStateException;
  public boolean getRollbackOnly () throws IllegalStateException;
  public boolean setRollbackOnly () throws IllegalStateException;
```

I metodi della EJBContext benché generici sono molto importanti: ad esempio getEJBObject() restituisce un reference remoto al bean object, che permette di essere utilizzato dal client o da un altro bean. I metodi della EntityContext e della SessionContext invece sono stati pensati per consentire una migliore gestione del ciclo di vita, della sicurezza e gestione delle transazioni di un entity o di un session.

Dal punto di vista del bean il contesto può essere utilizzato per avere un riferimento a se stesso tutte le volte che si deve effettuare una chiamata in loopback, ossia quando un bean invoca un metodo di un altro bean passando come parametro un reference di se stesso: questa operazione infatti non è permessa tramite la keyword this, ma tramite un reference remoto

ricavato dal contesto. I bean a tal proposito definiscono un metodo getObject() nella interfaccia EntityContext, il cui funzionamento è esattamente lo stesso. Ad esempio:

```
public class StudentBean extends EntityBean{
  public EntityContext context;

  public myMethod(){
    AnotherBean anotherBean  = ...
    EJBObject obj = Context.getObject() ;
    StudentBean studentBean;
    studentBean  = (StudentBean)PortableObject.narrow(obj, StudentBean.class);
    anotherBean.aMethod(studentBean);
  }
}
```

Il metodo getEJBHome(), disponibile sia per i session che per gli entity bean, è definito nella EJBContext: tale metodo restituisce un reference remoto al bean, e può essere utilizzato sia per creare nuovi bean, sia per effettuare delle ricerche nel caso di entity bean.

Durante il ciclo di vita all'interno del suo contesto, le varie informazioni accessibili tramite l'EJBContext possono variare. È per questo che tutti i metodi possono generare una IllegalStateException: ad esempio perde di significato ricorrere alla chiave primaria quando un bean si trova in stato di swapped, ossia quando non è assegnato a nessun EJBObject, anche se può disporre di un EJBContext.

Il getCallerPrincipal(), che dalla versione 1.1 sostituisce il getCallerIdentity(), permette di ricavare il reference al client invocante.

Quando invece si deve implementare un controllo a grana più fine sulla sicurezza, il metodo isCallerInRole() permette di controllare in modo veloce ed affidabile il ruolo di esercizio del client invocante.

Per quanto riguarda i metodi transazionali invece, essi saranno affrontati in seguito, quando si parlerà in dettaglio della gestione delle transazioni.

Il deploy di applicazioni EJB

Dopo aver visto tutte le varie parti che compongono un entity bean, resta da vedere come sia possibile rendere funzionante tale componente, ossia come effettuare il deploy di un EJB nel server. Le cose sono molto differenti a seconda che si desideri seguire la specifica 1.0 (che ormai è da considerarsi obsoleta) o 1.1: nel secondo caso le cose si sono semplificate moltissimo rispetto al passato, dato che è sufficiente scrivere in un documento XML tutte le informazioni necessarie. I file di deploy in formato XML sono essenzialmente due: il file standard (ejb-jar.xml) e quello proprietario (che prende il nome dell'application server utilizzato).

Il primo deve essere sempre presente e contiene tutte le informazioni necessarie per definire il comportamento di base del bean; il secondo invece contiene alcune informazioni specifiche della applicazione non sempre necessarie nei casi più semplici.

Di seguito sono riportate alcune parti del file standard facente riferimento alla applicazione SchoolManager in cui gli entity sono stati utilizzati come CMP 2.0.

La prima parte contiene una intestazione generica della applicazione:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
  "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
```

Ecco la porzione di XML dedicata alla definizione dell'entity bean Student:

```
<enterprise-beans>
  <entity>
    <display-name>Student</display-name>
    <ejb-name>Student</ejb-name>
    <home>com.mokabyte.mokabook2.ejb.schoolmanager.StudentHome</home>
    <remote>com.mokabyte.mokabook2.ejb.schoolmanager.StudentRemote</remote>
    <ejb-class>com.mokabyte.mokabook2.ejb.schoolmanager.StudentBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.String</prim-key-class>
    <reentrant>False</reentrant>
    <cmp-version>2.x</cmp-version>
    <abstract-schema-name>Student</abstract-schema-name>
    <cmp-field>
      <field-name>address</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>age</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>birthday</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>email</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>firstName</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>lastName</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>id</field-name>
```

```
    </cmp-field>
    <cmp-field>
      <field-name>password</field-name>
    </cmp-field>
    <primkey-field>id</primkey-field>
    <query>
      <query-method>
        <method-name>findAll</method-name>
        <method-params />
      </query-method>
      <ejb-ql>SELECT OBJECT(s) FROM Student s</ejb-ql>
    </query>
    <query>
      <query-method>
        <method-name>findByFirstName</method-name>
        <method-params>
          <method-param>java.lang.String</method-param>
        </method-params>
      </query-method>
      <ejb-ql>
          SELECT Object(s) FROM Student as s WHERE (LOCATE(?1, s.firstName) &gt;0)
       </ejb-ql>
    </query>
    <query>
      <query-method>
        <method-name>findByTeacherId</method-name>
        <method-params>
          <method-param>java.lang.String</method-param>
        </method-params>
      </query-method>
      <ejb-ql>
        SELECT OBJECT(s) FROM Student AS s, IN (s.studentClass.teachers) AS t WHERE t.id=?1
      </ejb-ql>
    </query>
  </entity>
```

Ecco la porzione dedicata alla definizione del session bean SchoolManager:

```
<session>
  <display-name>SchoolManager</display-name>
  <ejb-name>SchoolManager</ejb-name>
  <home>com.mokabyte.mokabook2.ejb.schoolmanager.SchoolManagerHome</home>
  <remote>com.mokabyte.mokabook2.ejb.schoolmanager.SchoolManagerRemote</remote>
```

```
    <ejb-class>com.mokabyte.mokabook2.ejb.schoolmanager.SchoolManagerBean</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>
  </session>
```

Ecco infine la parte dedicata alla definizione del comportamento transazionale dell'entity bean Student:

```
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>Student</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
```

e del session SchoolManager:

```
  <container-transaction>
    <method>
      <ejb-name>SchoolManager</ejb-name>
      <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  <container-transaction>
```

Si noti come tramite appositi tag XML siano definite le varie caratteristiche sia dell'entity bean, sia del session e di come si sia potuto specificare il comportamento transazionale e la gestione della persistenza dei vari campi.

Persistenza del bean e interfacciamento con il database

Per poter gestire i dati relativi al bean memorizzati nel database, si deve utilizzare una connessione verso il database, connessione che può essere ottenuta direttamente tramite le tecniche standard JDBC (soluzione sconsigliata e di fatto proibita dalla specifica), oppure ricavata per mezzo di JNDI da una sorgente dati opportunamente predisposta.

Utilizzando questa seconda soluzione, si potrebbe definire nel bean un metodo getConnection() in modo da utilizzarlo negli altri metodi del bean per poter leggere e scrivere i dati nel database; potrebbe essere così definito:

```
  private Connection getConnection() throws SQLExcepion {
```

```
    try{
      Context context= new InitialContext();
      DataSource source= (DataSource)context.lookup(SourceName);
      return source.getConnection();
    }
    catch(NamingException ne){
      throw new EJBException(ne);
    }
  }
```

dove SourceName è il nome JNDI dato alla sorgente dati: ogni bean infatti può accedere ad una serie di risorse definite all'interno del JNDI Environment Naming Context (ENC), il quale viene definito al momento del deploy del componente tramite il cosiddetto deployment descriptor. Questa organizzazione, introdotta con la versione 1.1 della specifica, è valida non solo per le sorgenti JDBC, ma anche per il sistema JavaMail o JMS: di fatto rende il componente ulteriormente indipendente dal contesto di esecuzione, rimandando al momento del deploy la sua configurazione tramite semplici file XML editabili con il tool di deploy utilizzato.

Ad esempio, supponendo che sia

```
SourceName = "java:comp/env/jdbc/myDb";
```

allora nel file deployment descriptor XML si avrebbe

```
...
<resource-ref>
  <description>DataSource per il MyBean</description>
  <res-ref-name>jdbc/myDb</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
  ...
```

Ciclo di vita di un entity bean

Alla luce della approfondita analisi dedicata agli entity bean, sia CMP che BMP, può essere utile affrontare più in dettaglio il ciclo di vita di questi componenti: questo dovrebbe consentire di avere chiaro quale sia il significato dei vari metodi visti fino a questo punto, per comprendere come e quando i vari metodi siano invocati.

Stato di pre-caricamento

In questo stato il bean ancora non esiste come entità astratta, ma piuttosto come collezione di file .class e di descriptor file: devono essere forniti al server la primary key, la remote e la home interface, oltre a tutti i file generati in modo automatico dal deploy.

Pooled State

Nel momento della partenza del server, questo carica in memoria tutti i bean di cui sia stato effettuato correttamente il deploy, posizionandoli nel pool dei bean pronti. In tale fase viene creata un'istanza del componente utilizzando il metodo bean.newIstance(), il quale a sua volta invoca il costruttore del bean: come accade in RMI, anche in questo caso per il bean non deve essere specificato nessun costruttore, e l'invocazione avviene su quello di default.

In questa fase tutte le variabili d'istanza assumono il loro valore di default e il container assegna l'EntityContext tramite il metodo setEntityContext().

In questo momento il componente è disponibile per essere utilizzato quando ci sarà bisogno di servire le richieste del client. Tale situazione perdurerà fino a quando non verrà invocato un metodo di ricerca.

Nessun bean in questo caso è assegnato ad un EJBObject, e nessuno contiene informazioni significative.

Ready State

Il passaggio al Ready State può avvenire o per esplicita chiamata di un metodo di ricerca da parte del client o per creazione diretta per mezzo dell'invocazione del metodo create.

Nel secondo caso per prima cosa viene creato un EJBObject, lo skeleton dell'oggetto secondo la terminologia RMI, gli viene assegnato un bean prelevato dal pool. In questo caso l'invocazione del metodo create da parte del client, viene propagata all'ejbCreate(): quando termina, una chiave primaria viene generata ed assegnata all'EJBObject. In questo momento sono effettuate tutte le operazioni di preparazione dei dati sul database in uno dei due modi, a seconda della politica scelta per la gestione della persistenza (CMP/BMP). Successivamente il controllo passa al metodo ejbPostCreate() per terminare la fase di inizializzazione.

Infine il client riceve lo stub dell'oggetto remoto, con il quale potrà effettuare tutte le invocazioni da remoto in perfetto accordo con quanto avviene nel modello RMI.

Leggermente differente è invece il caso relativo al passaggio al ready state tramite attivazione. In questo caso il bean viene prelevato dal pooled state dove era finito in seguito ad una passivazione: benché il bean non fosse presente in memoria, il container aveva mantenuto un legame fra lo stub del client e l'EJBObject: durante l'attivazione quindi i dati sono prelevati dal database e rassegnati al bean appena creato prima che questo sia accoppiato con l'EJBObject.

Analogamente il passaggio inverso, ossia dal ready al pooled può avvenire al verificarsi di vari eventi: sia su invocazione diretta del client sia in conseguenza di una passivazione attuata dal server. Anche in questo caso i metodi ejbStore() ed ejbPassivate() sono invocati in modo da mantenere sincronizzato il bean con i record del database. La sequenza precisa con cui tali invocazioni sono effettuate dipende dalla implementazione del server.

I session bean

Dovendo dare una definizione concisa ed esauriente di session bean si potrebbe dire che si tratta di componenti remoti manipolabili dal client secondo le ormai note interfacce home e remote in maniera analoga agli entity bean, rispetto ai quale differiscono per una caratteristica

fondamentale: mentre gli entity possono essere considerati come una rappresentazione di una struttura dati, i session incorporano al loro interno delle funzionalità o servizi. Per questo a volte si dice che inglobano la business logic del client spostandola sul server remoto. In definitiva, quindi, il client potrà utilizzare entity bean per accedere a dati remoti, ed invocare i servizi di uno o più session bean per manipolare tali dati.

Esistono due tipi di session bean, gli stateless e gli stateful, molto diversi fra loro per scopo e funzionamento: i primi sono componenti senza stato, mentre i secondi consentono di memorizzare uno stato fra due invocazioni successive dei metodi remoti da parte del client.

Stateless bean

Uno stateless bean non offre nessun mantenimento dello stato delle variabili remote fra due invocazioni successive dei metodi del bean da parte del client. L'interazione che si crea quindi fra il client e il bean è paragonabile a ciò che accade fra un web browser e un componente CGI server side: anche in quel caso infatti non vi è nessuna memorizzazione dello stato, se non tramite cookie o altre tecniche analoghe.

Gli stateless non dispongono quindi di alcun meccanismo atto a tenere traccia dei dati relativi ad un determinato client: questo è vero esclusivamente per le variabili remote, non per quelle di istanza interne alla classe. Ad esempio è possibile tramite una variabile di istanza realizzare un contatore del numero di invocazioni di un determinato metodo: ad ogni invocazione, ogni metodo potrà incrementare tale contatore anche se non è possibile ricavare l'identità del client invocante.

Tale separazione fra client e bean remoto impone quindi che tutti i parametri necessari al metodo per svolgere il suo compito debbano essere passati dal client stesso al momento dell'invocazione.

Un bean di questo tipo non ha quindi nessun legame con il client che lo ha invocato: durante il processo di swapping dei vari oggetti remoti, diverse istanze potranno essere associate ad un client piuttosto che ad un altro. Uno stateless quindi è spesso visto a ragione come un componente che raccoglie alcuni metodi di servizio.

Gli stateless offrono una elevata semplicità operativa ed implementativa, unitamente ad un buon livello di prestazioni.

Invocazione di metodi

Il meccanismo di base con cui sono invocati metodi remoti di un session è molto simile a quanto visto fino ad ora per gli entity. La differenza più lampante e forse più importante è quella legata alla diversa modalità con cui il bean può essere ricavato da remoto. Il metodo `getPrimaryKey()` infatti genera una `RemoteException` dato che un session non possiede una chiave primaria. Con la specifica 1.0 questo dettaglio era lasciato in sospeso, ed alcuni server potevano restituire un valore `null`.

Il metodo `getHandle()` invece restituisce un oggetto serializzato che rappresenta l'handle al bean, handle che potrà essere serializzato e riutilizzato in ogni momento: la condizione stateless infatti consente di riferirsi genericamente ad un qualsiasi bean del tipo puntato dall'handle. Per accedere all'oggetto remoto associato al bean si può utilizzare il metodo `getObject()` della interfaccia `Handle`

```
public interface javax.ejb.Handle{
   public abstract EJBObject getEJBObject() throws RemoteException;
}
```

Comportamento opposto è quello degli stateful, dove un handle permette di ricavare l'istanza associata al componente per il solo periodo di validità del bean all'interno del container.

Se il client distrugge esplicitamente il bean, tramite il metodo remove(), o se il componente stesso supera il tempo massimo di vita, l'istanza viene distrutta, e l'handle diviene inutilizzabile. Una successiva invocazione al metodo getObject() genera una RemoteException.

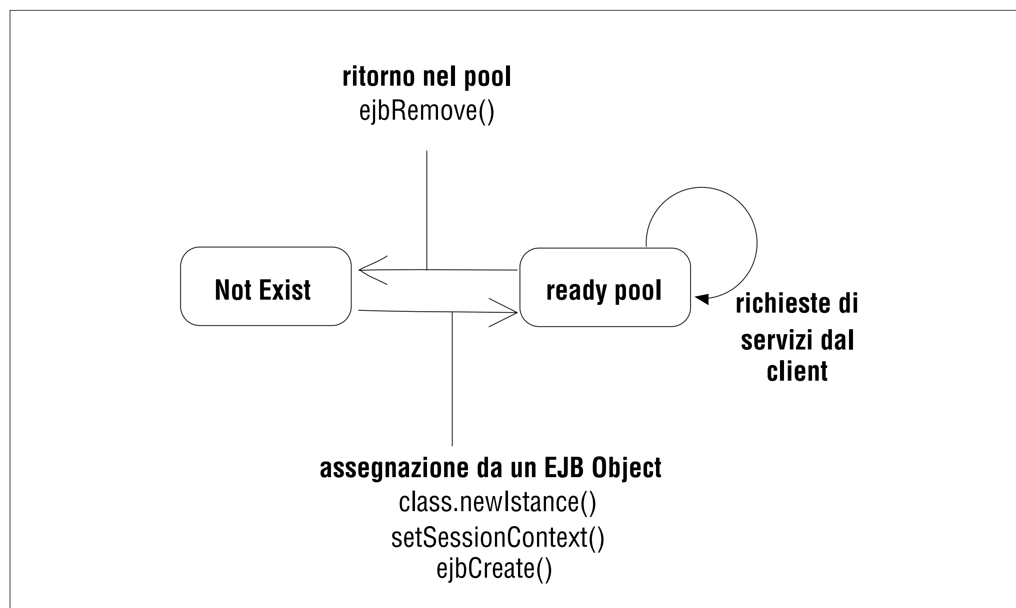
Ciclo di vita di uno stateless bean

Dato che uno stateless bean non rappresenta una particolare astrazione dati, né mantiene uno stato fra due invocazioni successive del client, il ciclo di vita per questi componenti risulta essere molto semplificato.

Gli stati che lo compongono sono solamente due, ed il passaggio da uno all'altro coinvolge un numero molto ridotto di operazioni: il primo, *not-exist state*, corrisponde alla non esistenza di alcun componente all'interno del container. Il secondo stato è invece il *ready-pool state* e corrisponde allo stato in cui il bean è pronto per l'invocazione da parte del client.

Un bean entra nel ready-pool su richiesta del container al momento del bisogno: molti server pre-istanziano un numero arbitrario di componenti al momento della inizializzazione del

Figura 8.9 – *Ciclo di vita di un session bean stateless.*



container, prima che un qualsiasi client ne richieda l'utilizzo; quando il numero dei bean istanziati risulta essere insufficiente, il server ne istanzia altri. Viceversa, quando il numero dei bean allocati è superiore alle reali necessità, il container rimuove dal ready-pool uno o più bean senza effettuare una particolare distinzione su quali eliminare, visto che tutti i componenti sono equivalenti fra loro.

Il passaggio verso il ready-pool avviene secondo una procedura ben precisa, che vede tre distinte fasi: per prima cosa il bean viene istanziato per mezzo del metodo Class.newIstance() corrispondente ad una specie di istanziazione statica del componente.

Successivamente tramite il metodo SessionBean.setSessionContext() al bean viene associato un contesto (classe EJBContext), che verrà mantenuto per tutto il suo periodo di vita.

Dato che uno stateless bean non prevede nel suo ciclo di vita lo stato di passivazione su memoria secondaria, il context può essere memorizzato indifferentemente in una variabile persistente o non persistente; la direttiva Sun invita i vari costruttori alla prima soluzione.

Alla terminazione del metodo ejbCreate() si considera terminato il processo di creazione del componente. Tale metodo viene invocato una sola volta durante il ciclo di vita del componente in corrispondenza del passaggio nello stato di ready-pool.

Quindi nel momento in cui il client invoca il metodo create() della home interface non si ha nessuna ripercussione sul componente che esiste già nello stato di ready-pool: non si ha l'invocazione del metodo ejbCreate() e si passa direttamente alla creazione di una istanza sul client.

Quando il bean si trova nello stato ready-pool è pronto per servire le richieste dei vari client: quando uno di questi invoca un metodo remoto dell'EJBObject relativo, il container associa alla interfaccia remota un bean qualsiasi prelevato dal pool per tutto il periodo necessario al metodo per svolgere il suo compito.

Al termine della esecuzione il bean viene disassociato dall'oggetto EJB, cosa nettamente differente sia rispetto agli entity bean che agli stateful, dove il bean resta associato allo stesso EJBObject per tutto il tempo in cui il client ha bisogno di interagire con il bean: questo fatto si ripercuote positivamente sulle prestazioni complessive come sulla quantità di memoria occupata dal sistema.

Sempre a causa della mancanza negli stateless di un meccanismo di persistenza dei dati, non verranno mai invocati i metodi in callback ejbActivate() ed ejbPassivate().

In definitiva il processo di istanziazione di un bean di questo tipo è molto più semplice rispetto agli altri casi: quello che però non cambia è la modalità con cui il client ricava un reference remoto. Ad esempio:

```
Object obj = jndiConnection.lookup(jndi_bean_name);
MyBeanHome myBeanHome =(MyBeanHome) PortableRemoteObject.narrow(obj, MyBeanHome.class);
MyBean myBean = myBeanHome.create();
```

Il passaggio inverso, da ready-pool a not-exist, avviene quando il server non necessità più della istanza del bean ossia quando il numero di bean nel pool è sovradimensionato rispetto alle necessità del sistema.

Il processo comincia dall'invocazione del metodo ejbRemove(), al fine di consentire tutte le operazioni necessarie per terminare in modo corretto (come ad esempio chiudere una connessione verso una sorgente dati).

L'invocazione da parte del client del metodo remove() invece non implica nessun cambiamento di stato da parte del bean, ma semplicemente rimuove il reference dall'EJBObject, che tra le altre cose comunica al server che il client non necessita più del reference.

È quindi il container che effettua l'invocazione dell'ejbRemove(), ma solamente al termine del suo ciclo di vita. Durante l'esecuzione di tale metodo il contesto è ancora disponibile al bean, e viene rilasciato solo al termine di ejbRemove().

Stateful bean

Questi componenti sono una variante del tipo precedente e dal punto di vista del ciclo di vita possono essere considerati una soluzione intermedia fra gli stateless bean e gli entity.

La definizione che descrive i session come oggetti lato server funzionanti come contenitori della business logic del client, è sicuramente più adatta al caso degli stateful che non degli stateless (per i quali è forse più corretto il concetto di *service components*).

Questo significa che un determinato bean servirà per tutta la sua vita lo stesso client, anche se il server manterrà sempre attivo un meccanismo di swap fra le varie istanze virtuali dello stesso bean.

Essendo uno stateful dedicato a servire uno e sempre lo stesso client, non insorgono problemi di accesso concorrente.

Quindi la differenza che sussiste fra un stateless ed un stateful è che il primo è molto vicino ad essere una raccolta di metodi di servizio, mentre il secondo invece rappresenta l'agente sul server del client.

Ciclo di vita di uno stateful bean

Nel ciclo di vita di uno stateful non è presente nessun meccanismo di pool del bean: quando un componente viene creato ed assegnato al client, continua a servire lo stesso client per tutto il suo periodo di vita.

Le diverse implementazioni dei server in commercio utilizzano però tecniche di ottimizzazione effettuando swapping in vari modi, anche se in modo del tutto trasparente, dato che in tal senso nella specifica ufficiale non è presente nessuna indicazione.

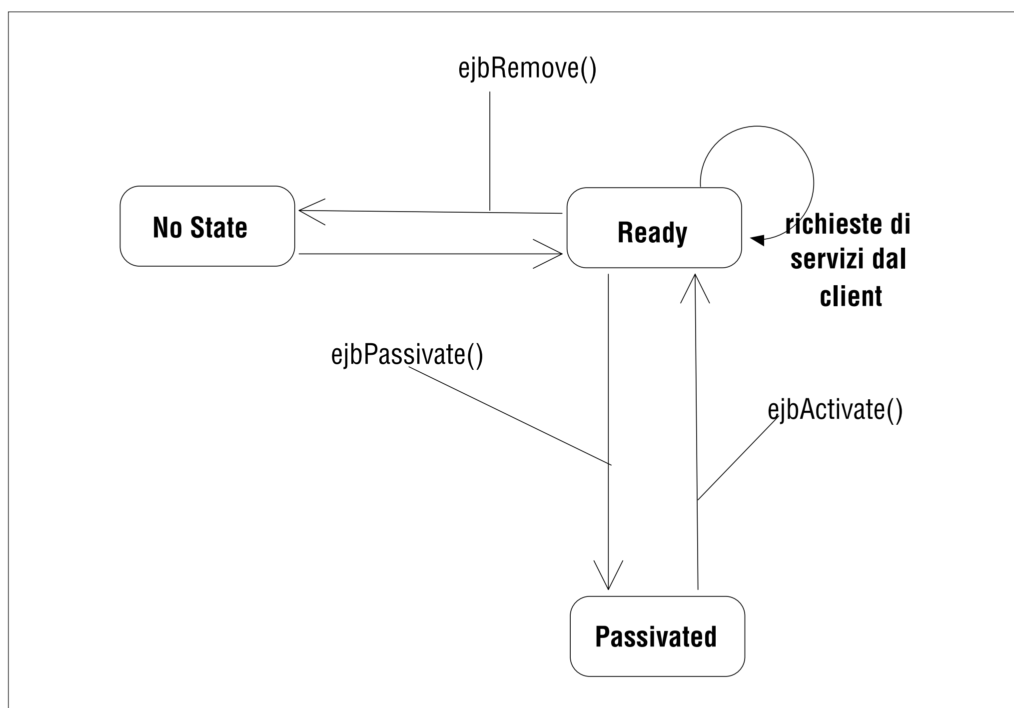
Il legame diretto e permanente che si instaura fra il client e l'EJBObject, in realtà viene virtualizzato quando, durante i lunghi periodi di inattività, il bean viene dereferenziato ed eliminato dal garbage collector. Per questo motivo è fornito un meccanismo di persistenza (salvataggio o passivazione e caricamento da memoria o riattivazione) al fine di mantenere il cosiddetto conversational state fra il client ed il bean.

Il bean ha la percezione del passaggio da uno stato ad un altro solo se implementa l'interfaccia SessionSynchronization, che fornisce un set di metodi di callback per la notifica degli eventi al bean, cosa particolarmente utile nel momento in cui si debbano gestire le transazioni.

Gli stati del ciclo di vita di uno stateful sono tre: *not-exist*, *ready* e *passivated*. Il primo è praticamente identico al caso degli stateless e corrisponde allo stato precedente alla istanziazione quando il bean è rappresentato da un insieme di file sul file system.

Quando il client invoca il metodo create() della home interface, il container comincia la fase di creazione del componente dando inizio al suo ciclo di vita.

Figura 8.10 – *Gli stati del ciclo di vita di uno stateful sono tre: not exist, ready e passivated.*



Per prima cosa il server crea un'istanza del bean tramite il metodo newIstance() e gli assegna un contesto tramite il setSessionContext(). Infine il container invoca il metodo ejbCreate() al termine del quale il componente, pronto per essere utilizzato, passa nello stato di ready.

Durante questa fase il componente è libero di rispondere alle invocazioni da parte del client, di accedere ad esempio a risorse memorizzate su database, o di interagire con altri bean.

Il passaggio nello stato di passivated può avvenire dopo un periodo più o meno lungo di inutilizzo da parte del client. In questo caso viene rimosso dalla memoria principale (ready state) e reso persistente tramite un qualche meccanismo dipendente dal server (ad esempio tramite serializzazione).

Tutto ciò che non dovrà essere reso persistente, in particolare le variabili transient o non serializzabili, dovrà essere messo a null prima della memorizzazione del bean, al fine di impedirne una errata invocazione da parte del client.

Infine se il componente supera il tempo di inattività massimo (time out) durante lo stato di passivated, semplicemente verrà eliminato così come ne verranno eliminati tutti i riferimenti nella memoria.

Il passaggio di stato inverso, da passivated a ready, avviene quando un client effettua una invocazione di un metodo remoto di un bean reso persistente.

In questo caso il componente viene deserializzato e reso disponibile: anche in questo caso si tenga presente che normalmente tutte le variabili primitive numeriche sono inizializzate a zero, mentre le altre non serializzabili a null. In questo caso la specifica lascia del tutto in sospeso questo aspetto (lasciato quindi al costruttore del server), per cui è bene utilizzare il metodo ejbActivate() per effettuare una corretta inizializzazione delle variabili della classe.

La gestione delle transazioni

L'accesso concorrente da parte di più client sullo stesso set di bean, e quindi ai dati memorizzati nel database, rende necessario un qualche sistema di controllo dei dati. Visto che la filosofia di base di EJB è quella di semplificare il lavoro dello sviluppatore di bean, anche in questo caso il lavoro "sporco" verrà effettuato dal container, consentendo una volta di più al programmatore di concentrarsi solo sulla logica business del componente.

Introduzione alle transazioni: il modello ACID

L'obiettivo primario di un sistema transazionale è garantire che l'accesso concorrente ai dati non porti a configurazioni incoerenti sui dati stessi o sui risultati di tali operazioni. Per ottenere questo obiettivo in genere si fa riferimento al cosiddetto modello ACID, ossia una transazione deve essere atomica (*A*tomic), consistente, nel senso di coerente (*C*onsistent), isolata (*I*solated) e duratura (*D*urable).

Atomic indica che tutte le operazioni che costituiscono la transazione devono essere eseguite senza interruzioni; se per un qualche motivo una qualsiasi delle operazioni dovesse fallire, allora il motore transazionale dovrà ristabilire la configurazione originaria prima che la prima operazione della transazione sia stata eseguita. Nel caso di transazioni sui database, questo significa che ai dati dovranno essere rassegnati i valori iniziali precedenti all'inizio della transazione. Se invece tutte le operazioni sono state eseguite con successo, le modifiche sui dati nel database potranno essere effettuate realmente.

La consistenza dei dati è invece un obiettivo che si ottiene grazie al lavoro congiunto del sistema transazionale e dello sviluppatore: il sistema fa uso sia di sistemi atti a garantire atomicità ed isolamento, sia di controlli sulle relazioni fra le tabelle del database inseriti nel database engine. Lo sviluppatore invece dovrà progettare le varie operazioni di business logic in modo da garantire la consistenza ossia l'integrità referenziale, la correttezza delle chiavi primarie, e così via.

L'isolamento garantisce che la transazione verrà eseguita dall'inizio alla fine senza l'interferenza di elementi esterni o di altri soggetti. La durevolezza infine deve garantire che le modifiche temporanee ai dati debbano essere effettuate in modo persistente in modo da evitare che un eventuale crash del sistema possa portare alla perdita di tutte le operazioni intermedie.

Lo scope transazionale

Il concetto di scope transazionale è di fondamentale importanza nel mondo EJB, ed indica l'insieme di quei bean che prendono parte ad una determinata transazione. Il termine scope

Tabella 8.2 – *Gli attributi transazionali in EJB 1.0 e 1.1.*

Attributo transazionale	EJB 1.1 valore testuale	EJB 1.0 costanti predefinite
Not Supported	NotSupported	TX_ NOT_SUPPORTED
Supports	Supports	Tx_SUPPORTS
Required	Required	TX_REQUIRED
Requires new	RequiresNew	TX_REQUIRES_NEW
Mandatory	Mandatory	TX_MANDA TORY
Never (1.1)	Never	-
Bean Managed (1.0)	-	TX_BEAN_MANAGES

viene utilizzato proprio per dar risalto al concetto di spazio di esecuzione: infatti ogni volta che un bean facente parte di un determinato scope invoca i metodi di un altro o ne ricava un qualche riferimento, causa l'inclusione di quest'ultimo nello stesso scope a cui appartiene lui stesso: di conseguenza quando il primo bean transazionale prende vita, lo scope verrà propagato a tutti i bean interessati dalla esecuzione.

Come si avrà modo di vedere in seguito, anche se può essere piuttosto semplice seguire la propagazione dello scope monitorando il thread di esecuzione di un bean, la propagazione dello scope deve tener conto della politica definita durante il deploy di ogni singolo bean, dando così vita ad uno scenario piuttosto complesso. La gestione della transazionalità di un bean e quindi la modalità con cui esso potrà prendere parte ad un determinato scope (sia attivamente sia perché invocato da altri bean) può essere gestita in modo automatico dal container in base ai valore dei vari parametri transazionali, oppure manualmente nel caso in cui si faccia esplicitamente uso di un sistema sottostante come Java Transaction Api (JTA).

Nel caso in cui si voglia utilizzare il motore transazionale del container EJB, si può facilmente definire il comportamento del bean tramite gli attributi transazionali (tab. 8.2). Il valore di tali attributi è cambiato dalla versione 1.0 alla 1.1: se prima infatti si faceva riferimento a costanti contenute nella classe ControlDescriptor, con la specifica 1.1 si è passati a più comode stringhe di testo, cosa che permette di utilizzare file XML per la configurazione manuale del bean.

È possibile definire il comportamento sia per tutto il bean sia per ogni singolo metodo: questa possibilità sebbene più complessa ed a rischio di errori, permette un maggior controllo e potenza. In EJB 1.0 per poter impostare uno dei possibili valori transazionali era necessario scrivere del codice Java. In EJB 1.1 l'utilizzo di script XML rende tale procedura più semplice. Ad esempio, riconsiderando l'esempio visto in precedenza, si potrebbe scrivere:

```
<container-transaction>
  <method>
    <ejb-name>BMPMokaUser</ejb-name>
    <method-name>*</method-name>
  </method>
```

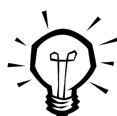
```
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
```

con il risultato di impostare tutti i metodi del bean BMPMokaUser al valore transazionale Required.

Normalmente, a meno di particolari esigenze, non è necessario né consigliabile gestire direttamente le transazioni: la capacità di poter specificare come i vari componenti possono prendere parte alle varie transazioni in atto è una delle caratteristiche più importanti del modello EJB, presente fin dalla specifica 1.0.

Significato dei valori transazionali

Si vedranno adesso i vari valori transazionali.



A volte si usa dire che una determinata transazione client è sospesa: con tale accezione si vuole significare che la transazione del client non è propagata al metodo del bean invocato, ma risulta temporaneamente sospesa fino a che il metodo invocato non termina. Si ricordi che per client si può intendere sia una applicazione standalone sia anche un altro bean.

Not Supported

Invocando all'interno di una transazione un metodo di un bean impostato con questo valore, si otterrà una interruzione della transazione; lo scope della transazione non verrà propagato al bean o ad altri bean da lui invocati. Appena il metodo invocato termina, la transazione riprenderà la sua esecuzione.

Supports

Nel caso in cui il metodo di un bean sia impostato a questo valore, l'invocazione da parte di un client incluso già in uno scope, provocherà la propagazione di tale scope al metodo. Ovviamente non è necessario che il metodo sia necessariamente invocato all'interno di uno scope, per cui potranno invocarlo anche client non facenti parte di nessuna transazione.

Required

In questo caso si ha la necessità della presenza di uno scope per l'invocazione del metodo. Se il client è coinvolto in una transazione lo scope verrà propagato, altrimenti ne verrà creato appositamente uno nuovo per il metodo del bean (scope che verrà terminato al termine del metodo).

Requires New

Il bean invocato entra in una nuova transazione, indipendentemente dal fatto che il client faccia parte o meno di una transazione. Se il client è coinvolto in una transazione, quest'ultima verrà interrotta fino al completamento della transazione del bean invocato. Il nuovo scope creato per il bean

verrà propagato esclusivamente a tutti i bean invocati dal bean di partenza. Quando il bean invocato terminerà la sua esecuzione, il controllo ritornerà al client che riprenderà la sua transazione.

Mandatory

Il bean deve sempre essere parte di una transazione; nel caso in cui il client invocante non appartenga a nessuno scope transazionale, il metodo del bean genererà una eccezione TransactionRequiredException.

Never (solo in EJB 1.1)

In questo caso il client invocante non può appartenere a nessun scope transazionale, altrimenti il bean invocato genererà una RemoteException.

Figura 8.11 – *Il funzionamento dell'attributo transazionale not supported. In questo caso lo scope della transazione non verrà propagato al bean o ad altri bean da lui invocati.*

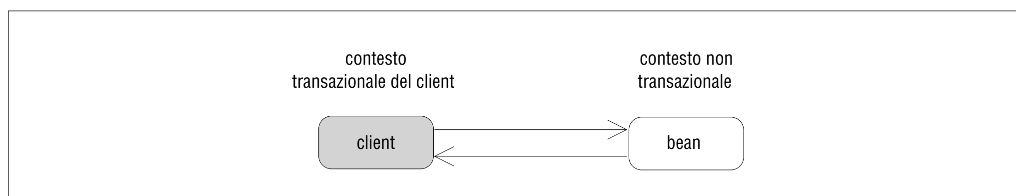


Figura 8.12 – *In questo caso il bean è in grado di entrare nello scope transazionale del client, anche se può essere invocato al di fuori di uno scope.*

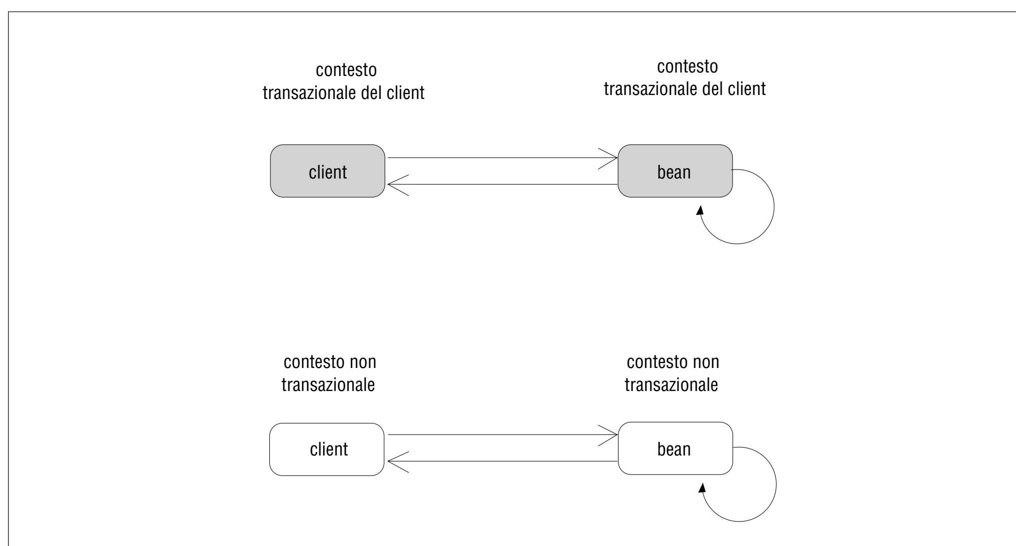


Figura 8.13 – *Un bean di questo tipo deve essere eseguito obbligatoriamente all'interno di uno scope: se il client opera in uno scope, il bean entrerà a far parte di quello del client; altrimenti un nuovo scope verrà creato appositamente per il bean.*

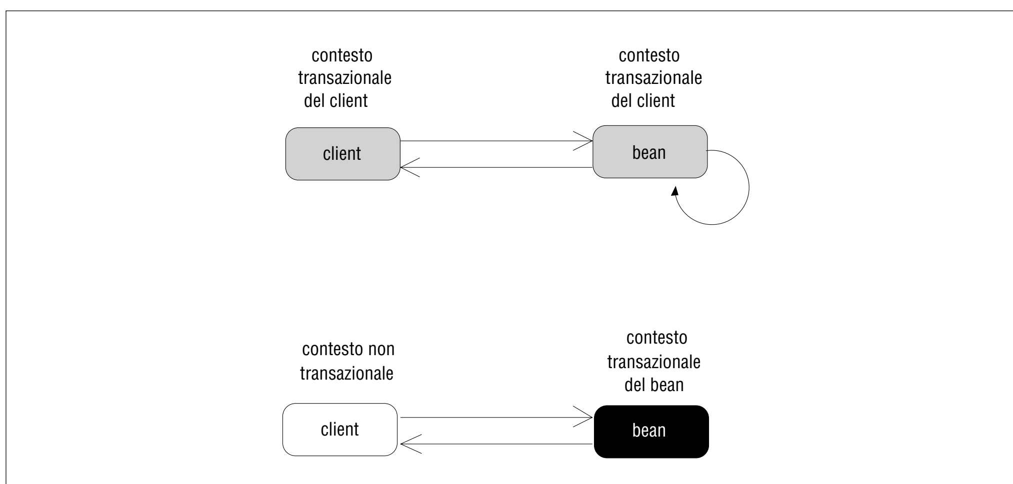


Figura 8.14 – *In questa configurazione il bean creerà sempre un suo nuovo scope.*

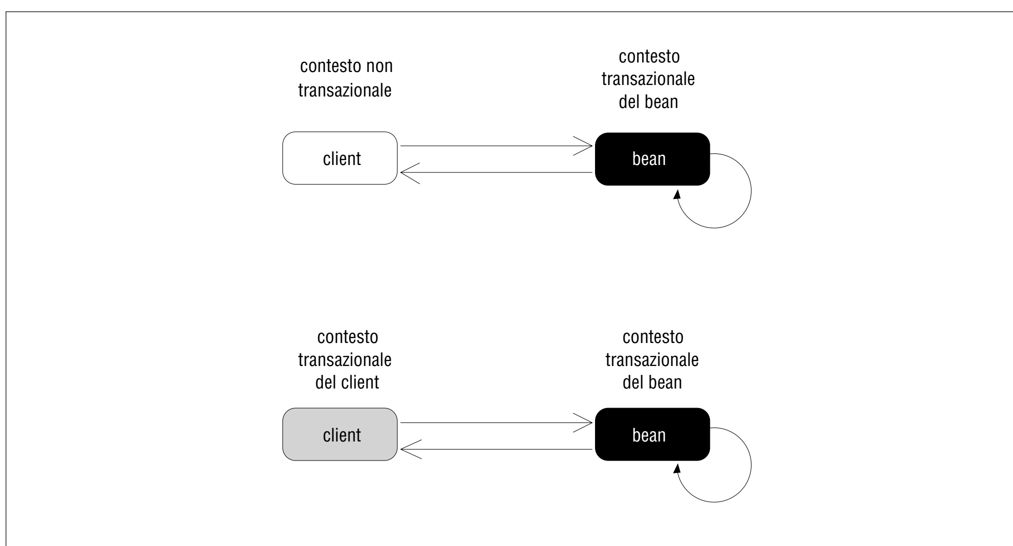


Figura 8.15 – *È una situazione simile alla required, anche se in questo caso la mancanza di uno scope preesistente nel client provoca la generazione di una eccezione da parte del bean, e non la creazione di uno scope apposito.*

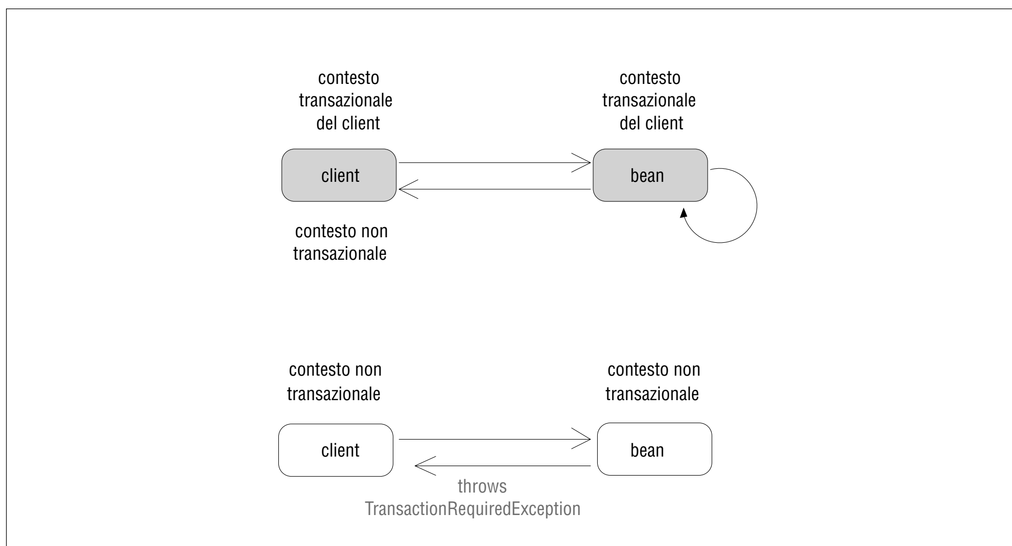


Figura 8.16 – *L'esatto contrario del caso precedente. In questo caso il bean non può assolutamente appartenere a uno scope, pena la generazione di una eccezione.*

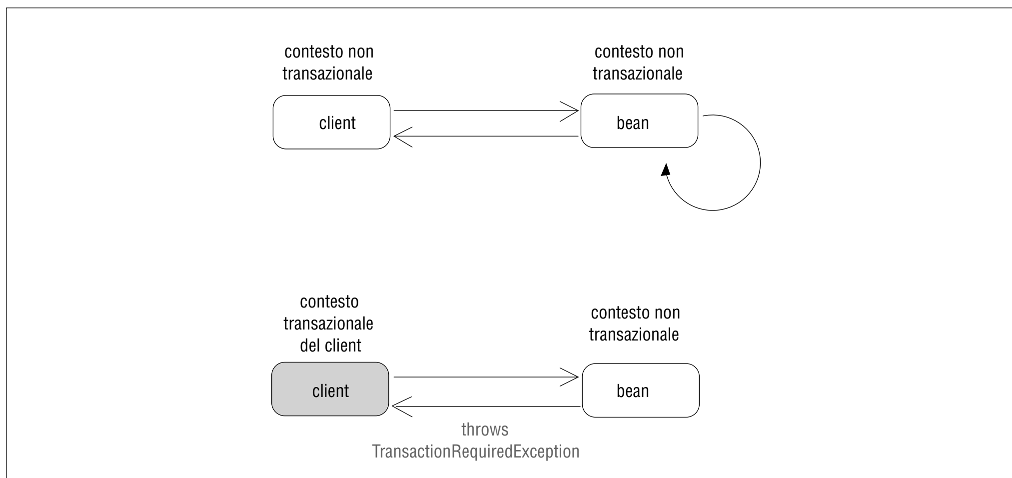
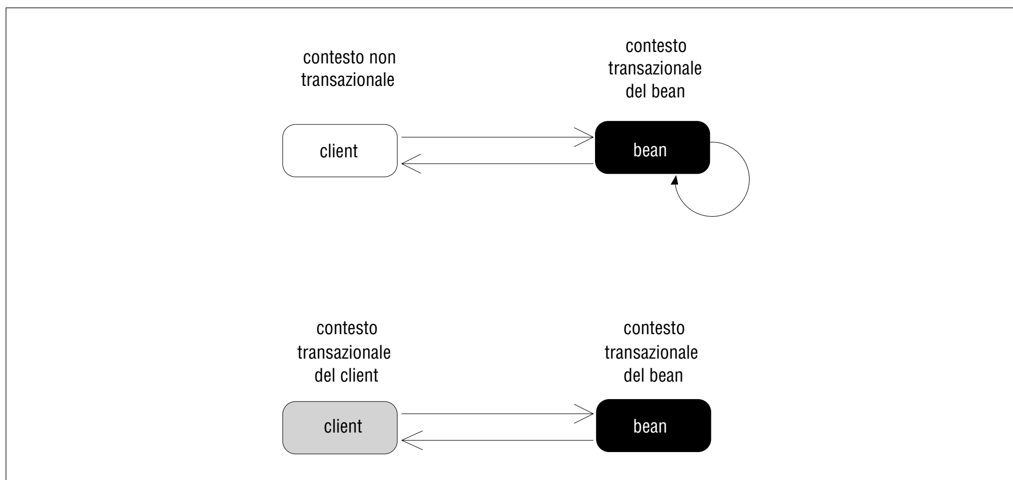


Figura 8.17 – *Il bean opererà sempre in un suo scope personale, gestito direttamente dal bean tramite un transaction engine come JTS.*



Bean Managed (solo in EJB 1.0)

In questo caso, possibile solo con la specifica 1.0, si indica che nessun supporto per la transazione deve essere specificato, dato che la gestione verrà effettuata manualmente (tramite ad esempio JTS) all'interno dei metodi del bean. I vari attributi possono essere impostati in modo autonomo sui vari metodi del bean, ad eccezione dell'ultimo caso, dove l'impostazione a bean managed per anche uno solo dei metodi obbliga a fornire il supporto manuale per tutti i metodi del bean.

Approfondimenti su database e integrità dei dati

Il concetto forse più importante e critico di un sistema transazionale concorrente è quello dell'isolamento (lettera I del modello ACID). Il livello di isolamento di una transazione in genere è valutabile in funzione del modo in cui riesce a risolvere i seguenti problemi:

- *Dirty reads*: si immagini il caso in cui due transazioni, una in lettura ed una in scrittura, debbano accedere ai medesimi dati. In questa situazione si possono avere incoerenze dei dati, nel caso in cui la transazione in lettura acceda ai dati appena modificati da quella in scrittura (qualora sia concessa la lettura contemporanea alla scrittura da parte di un altro metodo), quando quest'ultima dovesse per un motivo qualsiasi effettuare una rollback riponendo i dati nella configurazione originaria.

- *Repeteable reads*: questa condizione garantisce l'immutabilità dei dati al succedersi di differenti letture all'interno della stessa transazione (ossia elimina la possibilità che un soggetto possa modificare i dati oggetto della transazione mentre questa è in atto). Una

lettura non ripetibile si ha quando una transazione dopo una prima lettura, vedrà alla seguente le modifiche effettuate dalle altre transazioni. In genere questa condizione è garantita o tramite un lock sui dati, oppure tramite l'utilizzo di copie dei dati in memoria su cui effettuare le modifiche. La prima soluzione è probabilmente più sicura, anche se impatta pesantemente sulle prestazioni. La seconda invece può complicare molto la situazione, a causa delle difficoltà derivanti dalla necessità di sincronizzare i dati copia con quelli originali.

- *Phantom reads*: letture di questo tipo possono verificarsi quando nuovi dati aggiunti al database sono visibili anche all'interno di transazioni iniziate prima dell'aggiunta dei dati al database, transazioni che quindi si ritrovano nuovi dati la cui presenza è apparentemente immotivata.

Ecco alcune soluzioni comunemente utilizzate per prevenire i problemi di cui sopra:

- *Read locks*: questo blocco impedisce ad altre transazioni di modificare i dati quando una determinata transazione ne effettua una lettura. Questo previene l'insorgere di letture non ripetibili, dato che le altre transazioni possono leggere i dati ma non modificarli o aggiungerne di nuovi. Se il lock sia effettuato su un record, su tutta la tabella oppure su tutto database dipende dalla implementazioni particolare del database.

- *Write locks*: in questo caso, che si presenta tipicamente in operazioni di aggiornamento, alle altre transazioni viene impedito di effettuare modifiche ai dati; rappresenta un livello di sicurezza aggiuntivo rispetto al caso precedente, ma non impedisce l'insorgere di letture sporche dei dati (dirty reads) da parte di altre transazioni ed anche di quella in corso.

- *Exclusive locks*: questo è il blocco più restrittivo ed impedisce ad altre transazioni di effettuare letture o scritture sui dati bloccati: le dirty e phantom reads quindi non possono verificarsi.

- *Snapshot*: alcuni sistemi offrono un meccanismo alternativo ai lock, detto comunemente di snapshot (istantanea) dei dati: in questo caso sono create al momento dell'inizio della transazione delle vere e proprie copie dei dati, tali da permettere di lavorare in lettura e scrittura su una copia dei dati. Se questo elimina del tutto il problema dell'accesso concorrente, introduce problemi non banali relativamente alla sincronizzazione dei dati reali con le varie snapshot.

Livelli di isolamento delle transazioni

Per qualificare la bontà di un sistema transazionale in genere si fa riferimento ai cosiddetti livelli di isolamento.

- *Read Uncommitted*: una transazione può leggere tutti i dati uncommitted (ossia quelli ancora non resi persistenti), di altre transazioni in atto. Corrisponde al livello di garanzia più basso dato che può dar vita a dirty e phantom reads, cosi come possono verificarsi letture non ripetibili.

- *Read Committed*: una transazione non può leggere i dati temporanei (not committed) di altre transazioni in atto. Sono impedite le dirty reads, ma possono verficarsi le letture fantasma e le non ripetibili. I metodi di un bean con questo livello di isolamento non possono leggere dati affetti da una transazione.

- *Repeteable reads*: una transazione non può modificare i dati letti da un'altra transazione. Sono impedite le dirty reads e le letture fantasma ma possono verificarsi le letture non ripetibili.

- *Serializable*: corrisponde al livello massimo di sicurezza, dato che ogni transazione ha l'esclusivo diritto di accesso in lettura e scrittura sui dati. Si ha la garanzia contro le dirty reads, le letture fantasma e le non ripetibili.

Queste classificazioni corrispondono ai valori definiti come costanti in JDBC all'interno della `java.sql.Connection`. A parte il caso della gestione diretta all'interno del bean delle transazioni, la specifica 1.1 non prevede un meccanismo per l'impostazione tramite attributi del livello di isolamento come invece accadeva per la 1.0.

Scelta del livello di isolamento: il giusto compromesso

Al fine di assicurare in ogni istante la correttezza dei dati e delle operazioni svolte su di essi, si potrebbe pensare che la soluzione migliore possa essere l'adozione sempre e comunque di un livello di isolamento molto alto.

Come si è avuto modo già di accennare in precedenza questa non sempre è la soluzione migliore: è vera infatti la regola empirica che vede inversamente proporzionali il livello di isolamento e le performance complessive del sistema. Oltre ad un maggiore numero di operazioni di controllo da effettuare, l'accesso esclusivo in lettura, scrittura o entrambi trasforma gradualmente l'architettura da concorrente a sequenziale.

Purtroppo non esistono regole precise per poter scegliere in modo preciso e semplice quale sia la soluzione migliore da adottare in ogni circostanza e spesso ci si deve basare sull'esperienza valutando in base al contesto in cui si sta operando.

Ad esempio se si è in presenza di un bean a cui potenzialmente possano accedere contemporaneamente molti client (è il caso di bean che rappresentano una entità centrale, come la banca nell'esempio visto in precedenza), non sarà proficuo utilizzare un livello alto di isolamento, dato che questo metterà in coda tutte le chiamate di tutti i client.

È altresì vero che, proprio per l'elevato numero di client che accederanno al bean comune, si dovrà fornire un livello di sicurezza relativamente alto, dato che un piccolo errore si potrebbe ripercuotere su un numero elevato di applicazioni client.

La soluzione migliore potrebbe essere quella di separare i vari contesti e di applicare differenti livelli di isolamento per i vari metodi. Ad esempio tutti i metodi che dovranno essere invocati spesso per ottenere informazioni dal bean, ossia i metodi che corrispondono ad operazioni di lettura, potranno essere impostati con un livello basso di isolamento dato che la semplice lettura di un dato non è una operazione pericolosa. In questo caso il valore read uncommitted potrebbe andare bene.

Invece i metodi del tipo `setXXX` possono essere molto pericolosi ed in genere sono essere invocati molto di rado laddove certi elementi non cambino con facilità (per esempio, il nome di una banca). In questo caso deve essere fornito il livello massimo di isolamento visto che le prestazioni non sono un problema dato lo scarso uso dei metodi.

Ovviamente in tutti i casi intermedi si renderà necessaria una scelta adatta al caso specifico.

Gestione esplicita delle transazioni

Sebbene la gestione automatica delle transazione tramite il motore transazionale del server EJB sia la soluzione di gran lunga più semplice e preferibile, non è l'unica possibile: si possono infatti implementare tecniche manuali per ottenere risultati analoghi ma con un maggior controllo sulle singole operazioni effettuate.

A causa della complessità dell'argomento e delle conseguenze che tale scelta comporta la gestione manuale è raramente utilizzata: per questo motivo si affronteranno tali argomenti più per scopi didattici che per fornire realmente una conoscenza approfondita di un sistema transazionale. Si limiterà per quanto possibile la trattazione al caso della gestione delle transazioni in EJB, rimandando alla bibliografia chi volesse approfondire maggiormente gli argomenti relativi alla teoria transazionale.

La specifica EJB prevede che ogni server fornisca un supporto transazionale a livello di metodi del bean. Questa scelta offre già di per se una possibilità di configurazione molto dettagliata, e solo in rari casi si rende necessario scendere ad un livello di dettaglio maggiore.

Dato che la definizione del comportamento transazionale del bean avviene in fase di deploy, si può contare su una netta separazione fra il contesto transazionale e quello di business logic: ogni modifica effettuata in fase di deploy infatti permette di modificare radicalmente il comportamento o le prestazioni del bean senza la necessità di dover modificare la business logic del componente.

Bean non transazionali

Non necessariamente un bean deve lavorare in un contesto transazionale: si consideri ad esempio i session bean di tipo stateless, che possono essere visti come collezioni di metodi remoti di servizio; si prenda in esame il caso in cui il bean non effettui nessun tipo di accesso ai dati (ed operi al contempo solamente su variabili locali ai metodi).

Visto che un metodo di uno stateless può essere considerato in definitiva un servizio, mantenerlo fuori dal contesto transazionale (valore `Not Supported`) comporta un indubbio beneficio sulle prestazioni complessive.

La gestione esplicita delle transazioni

La gestione manuale, detta anche esplicita, delle transazioni si basa in genere su un qualche motore sottostante basato sul modello Object Transaction Model (OMT) definito da OMG: nel caso di Java ad esempio si potrebbe utilizzare il sistema JTS, che però per quanto potente offre una API piuttosto complessa e necessita di una chiara visione dell'obiettivo che si vuole raggiungere. Una soluzione sicuramente più semplice è quella offerta dalla Java Transaction Api (JTA), la quale essendo uscita dopo la versione 1.0 di EJB, non era disponibile inizialmente.

Dato che entrambi i sistemi fanno uso della interfaccia UserTransaction i pezzi di codice che si affronteranno saranno validi in entrambi i casi, a parte piccole modifiche di configurazione.

La API JTA è strutturata su due livelli, di cui il più alto e generico è quello normalmente utilizzato dal programmatore di applicazioni.

Per comprendere meglio quando possa essere realmente utile gestire manualmente le transazioni si prenda in esame il caso in cui un client debba eseguire le invocazioni dei due metodi remoti del bean prima di procedere e considerare realmente conclusa l'operazione. La transazione logica in questo caso è rappresentata dall'esecuzione di entrambi i metodi.

Appare quindi piuttosto evidente quale sia il vantaggio di questa soluzione: il client (al solito una applicazione o un altro bean) può effettuare un controllo diretto e preciso sui singoli passi della transazione, avendo sotto controllo l'inizio e la fine della stessa.

Indipendentemente dal tipo di client l'operazione da svolgere è la stessa, e si basa sull'utilizzo di un oggetto UserTransaction, anche se il modo per ottenere tale oggetto è piuttosto differente a seconda che il client sia una applicazione stand alone o un bean.

Con il rilascio della piattaforma J2EE Sun ha ufficialmente indicato come una applicazione debba ricavare una ricavare UserTransaction grazie a JNDI: quindi nel caso in cui la specifica di riferimento sia EJB 1.1 si potrebbe avere del codice di questo tipo:

```
Context jndiCtx = new InitialContext();
UserTransaction usrTrans;
usrTrans = UserTransaction)jndiCntx.looup("java:comp/UserTRansaction");
usrTrans.begin();
usrTrans.commit();
```

Fortunatamente nella maggior parte dei casi i vari produttori ricorrono ugualmente a JNDI per cui il codice precedente potrebbe diventare:

```
UserTransaction usrTrans;
usrTrans = (UserTransaction) jndiCntx.lookup("javax.trasaction.UserTransaction");
```

Detto questo è interessante notare che anche un bean può gestire le transazioni in modo esplicito. In EJB 1.1 i session bean cosiddetti *bean managed transaction* (in questo caso non è necessario specificare gli attributi transazionali per i metodi del bean) sono quei session bean il cui valore dell'attributo transaction-type sia impostato in deploy al valore bean: ad esempio

```
<ejb-jar>
```

```
<enterprise-bean>
  ...
<session>
  ...
  <transaction-type>Bean</transaction-type>
  ...
```

Invece gli entity non possono gestire la modalità "bean managed transaction".
Un bean per gestire la propria transazione deve ricavare al solito un reference ad un UserTransaction direttamente dall'EJBContext come ad esempio:

```
public class MyBean extends SessionBean{
  SessionContext ejbContext;
  public void myMethotd(){
    try{
      UserTransaction usrTrans;
      usrTrans = ejbContext.getUserTransaction();
      UsrTrans.begin();
      ...
      UsrTrans.commit();
  }
    catch(IllegalStateException ise){...}
    // gestione di altre eccezioni
    ...
  }
}
```

In questo caso si fa uso del contesto di esecuzione del bean per ricavare direttamente l'oggetto UserTransaction.

Nei session stateless bean una transazione gestita direttamente tramite l'oggetto UserTransction deve cominciare e terminare all'interno dello stesso metodo, dato che le stesse istanze dei vari bean in esecuzione sul server possono essere gestite in modo concorrente da più client.

Gli stateful invece, essendo dedicati a servire un solo client per volta, potranno avere transazioni che iniziano in un metodo e terminano in un altro. Nel caso in cui il client appartenente ad un determinato scope transazionale invochi su un metodo di un bean in cui sia effettuata una gestione diretta delle transazioni, si otterrà una sospensione della transazione del client fino a quando il metodo remoto del bean non abbia terminato; questo sia che la transazione del metodo del bean inizi all'interno del metodo stesso, sia che sia iniziata precedentemente all'interno di un altro metodo. Quest'ultima considerazione fa comprendere quanto sia importante evitare di utilizzare transazioni "spalmate" su più metodi, a causa della maggiore complessità che si introduce nel sistema.

Gestione delle transazioni dal punto di vista del server

Il server EJB per fornire il supporto transazionale ai vari bean deve fornire una implementazione delle interfacce UserTransaction e Status: non è necessario quindi che il server supporti il resto della API JTA né che sia utilizzato il sistema JTS.

L'interfaccia UserTransaction ha la seguente definizione

```
interface javax.transaction.UserTransaction {
   public abstract void begin();
   public abstract void commit();
   public abstract void rollback();
   public abstract int getStatus();
   public abstract void setRollbackOnly();
   public abstract void setTransactionTimeout(int secs):
}
```

Ecco il significato dei vari metodi e le funzioni che svolgono:

```
begin()
```

Fa partire la transazione ed associa il thread di esecuzione con la transazione appena creata. Possono essere generate eccezioni di tipo NotSupportedException (nel caso in cui il thread sia associato ad un'altra transazione) o di tipo SystemException nel caso in cui il transaction manager incontri un problema imprevisto.

```
commit()
```

Completa la transazione associata al thread il quale poi non apparterrà a nessuna transazione. Tale metodo può generare eccezioni di tipo IllegalStateException (nel caso in cui il thread non appartenesse a nessuna transazione iniziata in precedenza), oppure SystemException se come in precedenza dovessero insorgere problemi inaspettati. Una TransactionRolledBackxception verrà generata se la transazione viene interrotta o se il client invoca il metodo UserTransaction.rollBackOnly().

Nel caso peggiore verrà prodotta una HeuristicRollbackException ad indicare l'insorgere di una cosiddetta *decisione euristica*: questo particolare tipo di evento corrisponde alla decisione presa da uno qualsiasi degli elementi che prendono parte alla transazione senza nessuna autorizzazione né indicazione da parte del transaction manager, di effettuare una commit o una rollback. In questo caso la transazione perde ogni livello di atomicità e la consistenza dei dati non può essere in alcun modo essere considerata affidabile.

```
rollback()
```

Provoca una rollback della transazione. Una SecurityException verrà prodotta se il thread non è autorizzato ad effettuare la rollback; anche in questo caso verrà generata una IllegalStateException, se il thread non è associato a nessuna transazione.

```
setRollBackOnly()
```

Imposta la modalità rollback forzata, provocando obbligatoriamente la generazione di una rollback in ogni caso. Come in precedenza verrà generata una Illegal-StateException, se il thread non dovesse essere associato a nessuna transazione, una SystemException invece verrà lanciata se il transaction manager dovesse incontrare un problema imprevisto.

```
setTransactionTimeout(int secs)
```

Imposta il tempo massimo entro il quale la transazione debba essere conclusa. Se tale valore non viene impostato, il server utilizzerà quello di default che è dipendente dalla particolare implementazione. Una SystemException verrà lanciata se il transaction manager dovesse incontrare un problema imprevisto

```
getStatus()
```

Per chi volesse scendere ad un livello più dettagliato di controllo, tramite tale metodo è possibile ricevere un valore intero da confrontare con le costanti memorizzate nella classe Status la cui definizione è riportata di seguito

```
interface javax.transaction.Status{
  public final static int STATUS_ACTIVE;
  public final static int STATUS_COMMITTED;
  public final static int STATUS_COMMITTING;
  public final static int STATUS_MARKED_ROLLBACK;
  public final static int STATUS_NO_TRANSACTION;
  public final static int STATUS_PREPARED;
  public final static int STATUS_PREPARING;
  public final static int STATUS_ROLLED_BACK;
  public final static int STATUS_ROLLING_BACK;
  public final static int STATUS_UNKNOWN;
}
```

Il significato di tali valori dovrebbe essere piuttosto intuitivo, per cui non ci dilungheremo oltre nella loro analisi.

Considerazioni sulla gestione manuale delle transazioni

Come si è avuto modo di constatare la gestione diretta delle transazioni sicuramente rappresenta un meccanismo potente per controllare più nel dettaglio i vari aspetti della applicazione sia nella parte di business logic che relativamente alla parte di gestione del ciclo di vita del bean (metodi di callback). Tale potenza espressiva introduce un livello di complessità che in genere non trova giustificazione nella maggior parte dei casi: oltre a richiedere una maggiore cono-

scenza della teoria transazionale, questa soluzione va contro il principio di separazione fra business logic e motore transazionale, filosofia di base di EJB. Dato che il motore transazionale rappresenta probabilmente il cardine principale di tutto EJB, nel caso si opti per la gestione manuale potrebbe diventare un inutile complicazione utilizzare EJB solamente per usufruire dei servizi restanti.

L'esperienza insegna che nella stragrande maggioranza dei casi è meglio affidarsi ai sistemi offerti dal server EJB, eventualmente giocando in modo opportuno sulla configurazione dei vari bean.

La gestione delle eccezioni all'interno di un container EJB

Le eccezioni in Java sono organizzate in due principali categorie, le checked e le unchecked. Nell'ambito di EJB ed in particolare della gestione delle transazioni, è utile analizzare le cose sotto un punto di vista leggermente differente, distinguendo fra application exception e system exception (tutte quelle che derivano da RuntimeException RemoteException e sottotipi come le EJBException).

L'eccezione più importante in ambito EJB è la RemoteException. La regola base introdotta nella versione 1.0 della specifica EJB imponeva che tutti i metodi remoti dichiarassero il rilancio di questo tipo di eccezione; nel caso in cui all'interno dei vari metodi di callback o di business logic possano essere generate altri tipi di eccezioni, queste dovranno essere catturate e propagate verso l'esterno come eccezioni remote.

Questa tecnica, tipica nel mondo RMI, prende il nome di *exception wrapping*, ed è particolarmente utile in tutti quei casi in cui la business logic del bean si appoggia a sistemi sottostanti per la gestione di particolari risorse, come nel caso di JDBC, JNDI o JavaMail.

Un esempio molto semplice di utilizzo della exception wrapping potrebbe essere:

```
public void ejbLoad() throws RemoteException{
  ...
  try{
    ... Esegue una ipotetica operazione con una sorgente JDBC
  }
  catch(SQLException sqle){
    throw new RemoteException(sqle);
  }
}
```

Con il passaggio alla versione 1.1 di EJB, le eccezioni generate devono essere inglobate tramite wrapper in eccezioni di tipo EJBException: quest'ultima, discendendo dalla RuntimeException, non richiede la dichiarazione della clausola throws nella firma del metodo. Il pezzo di codice appena visto nella versione 1.1 potrebbe quindi essere riscritto nel seguente modo:

```
public void ejbLoad(){
  ...
  try{
    ... Esegue operazioni con una connessione JDBC
  }
  catch(SQLException sqle){
    throw new EJBException(sqle);
  }
}
```

Il wrapping dovrà essere attuato in modo da propagare una EJBException (o RemoteException) quando si sia verificato un problema in uno degli strati software sottostanti (JDBC, JNDI, JMS o altro), mentre si dovrà generare una eccezione proprietaria nel caso di un errore o problema nella business logic.

Questa regola vale per le checked exception, dove si può controllare l'eccezione tramite il costrutto try-catch. Per le eccezioni unchecked (che di fatto possono insorgere in ogni momento senza la possibilità di controllo da parte del programma), il metodo remoto propagherà sempre una eccezione remota di tipo RemoteException.

Application exception vs. system exception

Una eccezione applicativa è una qualsiasi eccezione che non estende la java.lang.RuntimeException o la java.rmi.RemoteException.

Sono invece eccezioni di sistema tutte quelle che derivano dalla java.lang.RuntimeException, compresa la EJBException.

Ogni volta che all'interno di un metodo viene lanciata una eccezione di sistema, l'eventuale transazione in corso viene annullata e viene eseguita una rollback di sistema. Le eccezioni applicative invece non causano una rollback della transazione in corso.

Questa semplice regola è alla base di tutto il meccanismo di EJB relativo alla gestione delle eccezione e della interazione con le transazioni.

System exception

Le eccezioni di sistema sono tutte quelle che derivano dalla RuntimeException, compresa quindi anche la EJBException.

Tutte le eccezioni di questo tipo non devono essere necessariamente dichiarate nella clausola throws nella firma del metodo, così come non è necessario inserirle all'interno di un blocco try–catch.

Se l'eccezione si verifica all'interno del metodo che ha iniziato la transazione allora la rollback verrà immediatamente effettuata dal container. Se invece il metodo è un metodo del client, allora la transazione verrà annullata e verrà propagata notifica al client invocante. Eccezioni di questo tipo sono gestite direttamente dal container che effettua le seguenti operazioni:

- Rollback della transazione;

- Log della eccezione per notificare eventualmente l'amministratore di sistema: la specifica in questo caso non definisce come la notifica debba essere eseguita, particolare che è lasciato alla particolare implementazione. JBoss ad esempio aggancia i messaggi di questo tipo ad un appender Log4J che poi viene configurato in modo opportuno dall'amministratore di sistema in modo da dirigere i messaggi verso una coda JMS, una casella di posta, o un file di log.

- L'istanza dell'EJB viene dereferenziata e mandata al garbage collector. In questo caso infatti si presume che lo stato mantenuto nel bean non sia coerente e quindi inutile per l'utilizzo.

All'interno di tale meccanismo non vi è differenza se le eccezioni si siano verificate all'interno di un metodo di callback (ejbStore(), ejbLoad(), e così via) o all'interno di un metodo di business logic (tipicamente quelli di un session).

Relativamente al terzo punto, l'eliminazione del bean, l'impatto di questo genere di operazione dipende molto dal tipo di bean: nel caso di session bean stateless ed entity, non essendo dedicate ad un particolare client, esse potranno essere rimosse dal container senza particolari conseguenze. Il client quindi in questo caso non percepisce questo evento.

Nel caso di session stateful invece l'impatto è molto più forte, dato che essi sono dedicati ad un singolo client e ne mantengono lo stato. In questo caso quindi la rimozione del bean provoca la distruzione del cosiddetto conversational state con il client: successive invocazioni da parte di questo provocano il lancio di una eccezione di tipo NoSuchObjectException, figlia della RemoteException. Infine nel caso di MDB una eccezione all'interno del metodo onMessage() o in uno dei metodo di callback, provoca la rimozione del bean.

Se la transazione è gestita dal bean (BMT), il messaggio potrebbe essere nuovamente recapitato a seconda di quando il container notifica il ricevimento del messaggio. Nel caso di CMT la transazione viene annullata ed il messaggio nuovamente inviato dal container a seconda della implementazione.

Il client di un session o di un entity riceve sempre una RemoteException: se il client ha iniziato la transazione, l'eccezione di sistema lanciata dal metodo del bean, viene wrappata all'interno di una TransactionRolledbackException in modo da avvertire in modo più esplicito il client che si è verificata una rollback.

Nel caso in cui siano utilizzate le interfacce locali per l'invocazione, in EJB 2.0 quindi, ai client viene notificata una EJBException. Anche in questo caso se è il client che ha iniziato la transazione, esso riceverà una TransactionRolledbackException. In tutti gli altri casi (sia CMT che BMT) l'eccezione viene ripropagata al client invocante tramite una EJBException.

Nel caso in cui l'eccezione si verifichi in un sottosistema gestito dal bean (SQLException o JMSException se i sottosistemi sono JDBC o JMS), si dovrebbe sempre generare una EJBException, anche se in genere il progettista decide di gestire questi casi anomali agendo in modo da far rientrare il problema, oppure notificando il client con un messaggio personalizzato, utilizzando quindi una eccezione applicativa. Questa scelta però deve essere fatta solo quando si ha completa conoscenza del funzionamento del sistema e delle ripercussioni derivanti dall'insorgere di una eccezione verificatasi nel sottosistema. In tutti i casi ambigui è bene usare sempre la soluzione di default basata sul rilancio di

EJBException. Si faccia attenzione che tutti i metodi di callback dichiaravano nella specifica 1.0 eccezioni di tipo RemoteException: questa soluzione è stata deprecata a partire dalla 1.1.

Application exception

Le eccezioni applicative in genere sono lanciate in concomitanza di errori di business logic, e sono sempre rinviate verso il client senza che ci sia alcun meccanismo di wrapping da parte del container. Normalmente non generano alcuna rollback ed il client quindi non ha possibilità di intervenire o di recuperare lo stato dopo una eccezione di questo tipo. Eccezioni di questo genere vengono utilizzate dal programmatore se ad esempio un metodo viene invocato ma non tutti i parametri sono stati forniti al bean, oppure se una determinata operazione non può essere eseguita perché lo stato del bean non è congruo con l'operazione stessa: in tal caso il proseguo delle operazioni porterebbe ad una eccezione di sistema, eventualità tipicamente più grave, su cui il programmatore non ha possibilità di intervenire.

Ad esempio il pagamento di un acquisto senza che sia stato fornito un numero di carta di credito valido, oppure un login senza che la password sia stata valorizzata. Per questo le eccezioni applicative possono essere considerate casi meno gravi e vengono spesso utilizzate come sistemi di notifica di messaggi particolari dallo strato di business logic verso il client. Se è questo lo scenario di utilizzo delle eccezioni applicative, tali controlli devono essere effettuati prima che nel metodo del bean sia eseguita una qualsiasi operazione transazionale da e verso il sottosistema, in modo che il client possa essere avvertito e semmai ritentare l'operazione remota fornendo tutte le informazioni ed i dati necessari, prima che i dati nel sistema cambino.

I metodi di business logic possono lanciare ogni tipo di eccezione applicativa, eccezioni che devono essere definite nelle firme dei metodi nelle interfacce remote e locali, così come nelle implementazioni dei metodi nella classe dell'EJB.

Anche i metodi di callback possono rilanciare alcune eccezioni definite in javax.ejb che sono dette standard application exception: CreateException DuplicateKeyException, FinderException, ObjectNotFoundException, RemoveException che sebbene fornite dal sistema sono considerate eccezioni applicative a tutti gli effetti e vengono rilanciate al client senza nessuna intermediazione da parte del container e senza re-wrapping sotto forma di RemoteException. Non necessariamente tali eccezioni provocano una rollback della transazione, offrendo al client la possibilità di ritentare l'operazione. Tali eccezioni sono lanciabili sia dal bean (nel caso ad esempio di entity CMP) sia anche direttamente dal container.

Standard application exception e loro significato

Di seguito è riportata una breve analisi delle eccezioni applicative standard. Tutte e quattro possono essere lanciate dal container se la persistenza è di tipo CMT, ma può essere anche uno dei metodi creazionali ejbCreate() o ejbPostCreate() a lanciarla esplicitamente.

CreateException

Viene generata dal metodo create() della interfaccia remota: può essere lanciata dal container. La sua generazione indica che si è verificato un errore grave che impedisce la creazione del

bean stesso (ad esempio parametri non validi o non completi). In un CMT se il container lancia questa eccezione non si ha la certezza del rollback della transazione e si deve in tal caso procedere in modo manuale ad effettuare i dovuti controlli.

Se l'integrità dei dati è un aspetto fondamentale dovrà essere annullata la transazione prima di lanciare questa eccezione.

DuplicateKeyExcetpion

È un sottotipo della precedente e viene lanciata all'interno del metodo ejbCreate(). Indica che la chiave primaria a cui è associato un entity esiste già nel database. L'entity non può essere creato.

FinderException

Questa eccezione viene lanciata dai metodi di ricerca presenti nella interfaccia home. Indica che la ricerca ha generato un errore per un motivo applicativo qualsiasi (argomenti non validi o altro). Questo evento deve essere associato solamente nel caso di errori di ricerca, non di oggetti non trovati (caso per il quale si deve utilizzare l'eccezione successiva).

ObjectNotFoundException

Questa eccezione deve essere lanciata se non viene trovato nessun record nel database corrispondente ai parametri di ricerca utilizzati. Più precisamente nel caso di metodi a ricerca multipla deve essere restituita una collection vuota, mentre per i metodi a ricerca singola deve essere rilanciata questa eccezione. Genericamente la transazione non viene annullata.

Non si dovrebbe utilizzare questa eccezione in tutti quei casi in cui si sia verificato un errore applicativo dovuto a motivi particolari, come nel caso precedente.

RemoveException

Questa eccezione viene generata dai metodi remove() della interfaccia locale e remota in corrispondenza di problemi all'atto della cancellazione. La transazione non è detto che sia annullata, e devono essere fatti controlli espliciti in tal senso.

EJB 2.0: CMP 2.0 e Abstract Persistent Model

Con l'avvento della specifica EJB 2.0 sono state introdotte alcune radicali modifiche relativamente agli entity bean di tipo CMP.

Anche se gli application server dovranno ancora supportare entrambe le specifiche CMP (1.1 e 2.0), i due sistemi sono incompatibili fra loro tanto che il programmatore dovrà scegliere quale dei due adottare. Le innovazioni riguardano tre aspetti principali:

- la possibilità di definire lo schema di persistenza del bean in maniera astratta, più potente e con una granularità più fine;
- la gestione automatica dei legami relazionali fra entity bean differenti;

- l'introduzione di EJBQL, un linguaggio standard per la definizione delle procedure di ricerca tramite script da inserire nel deployment file.

Concettualmente le prime due modifiche sono molto importanti, mentre il linguaggio di query, sebbene rappresenti un importante passo in avanti nel processo di standardizzazione, non costituisce una novità a tutti gli effetti dato che i vari produttori già implementavano soluzioni analoghe (ma essendo proprietarie limitavano fortemente la portabilità delle applicazioni da un application server all'altro).

Abstract Programming Schema

Così come avveniva con la versione 1.1, anche in CMP 2.0 tutto il processo di sincronizzazione dello stato di un entity viene gestito dal container. Adesso però il programmatore possiede alcuni strumenti in più per aumentare il livello di astrazione della applicazione rispetto al sistema sottostante di persistenza (container + database). In CMP 2.0 nasce il concetto di campo virtuale (*virtual field*), così chiamato perché, sebbene faccia parte a tutti gli effetti del set di attributi del bean, non è implementato direttamente dal programmatore ma dal container al momento del deploy.

Con il termine di campo virtuale, ci si riferisce quindi alla coppia di metodi set/get e non al campo in sé che non esiste prima del deploy. I campi virtuali si suddividono poi in *relazionali* (vera innovazione di CMP 2.0) e di *persistenza*.

L'insieme dei campi virtuali, ossia dei corrispondenti metodi astratti, danno luogo al concetto di Abstract Programming Model. Il corrispondente codice XML nel deployment descriptor dà vita al cosiddetto Abstract Programming Schema.

Adottare l'Abstract Programming Model significa trasformare gli entity in classi astratte le cui implementazioni verranno realizzate dal container al momento del deploy: tali classi concrete sono dette *classi di persistenza* ed includono in modo esplicito il codice relativo all'accesso al database per realizzare il processo di persistenza sui dati; tale codice è ottimizzato in funzione del tipo e del prodotto specifico di database scelto.

In analogia con quanto avviene con il mondo dei database relazionali, nel caso in cui si sia abilitata la cancellazione in cascata (tag <cascade-delete>), la rimozione di un entity partecipante ad una relazione con altri come elemento centrale, provoca la rimozione anche dei dependent object: ovviamente questo è consentito solo per le relazioni 1 a 1. Anche per questo motivo, per non alterare l'integrità delle relazioni, il client non dovrebbe accedere agli oggetti dipendenti.

A conferma o come ultima conseguenza delle indicazioni di cui sopra, la specifica impone che i campi di relazione debbano sempre essere reference delle interfacce locali (vedi oltre) degli oggetti dipendenti. Questo elimina ogni dubbio, trattandosi di reference locali, non possono essere passati ai client remoti.

Infine è da tenere presente che, per mantenere la retrocompatibilità, gli application server EJB 2.0 dovranno continuare a fornire il supporto con il modello dei CMP 1.1, anche se i due modelli sono incompatibili fra loro. Il programmatore potrà quindi scegliere quale soluzione adottare ma non potrà utilizzare entrambi i modelli contemporaneamente all'interno della stessa applicazione

La definizione della interfaccia remota di Student in questo caso è la seguente:

```
public interface StudentRemote extends EJBObject {
   public void addBooks(Collection books) throws RemoteException;
   public void assignCurriculum(String curriculumId) throws RemoteException;
   public void addTestResults(Collection testResults) throws RemoteException;
   public void addTestResult(TestResultLocal testResultLocal) throws RemoteException;
   public void addBook(BookLocal BookLocal) throws RemoteException;
   public void setAddress(String address) throws RemoteException;
   public String getAddress() throws RemoteException;
   public void setAge(String age) throws RemoteException;
   public String getAge() throws RemoteException;
   public void setBirthday(String birthday) throws RemoteException;
   public String getBirthday() throws RemoteException;
   public void setEmail(String email) throws RemoteException;
   public String getEmail() throws RemoteException;
   public void setFirstName(String firstName) throws RemoteException;
   public String getFirstName() throws RemoteException;
   public void setLastName(String lastName) throws RemoteException;
   public String getLastName() throws RemoteException;
   public void setPassword(String password) throws RemoteException;
   public String getPassword() throws RemoteException;
   public String getId() throws RemoteException;
}
```

I metodi astratti di get e set in questo caso danno luogo al cosiddetto Asbract Programming Model. Anche nella implementazione del bean tali metodi rimangono astratti: per brevità di seguito è riportata una porzione sintesi del bean StudentBean:

```
public abstract class StudentBean implements EntityBean {

   EntityContext entityContext;

   public void ejbRemove() throws RemoveException { }
   public abstract void setAddress(String address);
   public abstract void setAge(String age);
   public abstract void setBirthday(String birthday);
   public abstract void setEmail(String email);
   public abstract void setFirstName(String firstName);
   public abstract void setLastName(String lastName);
   public abstract void setPassword(String password);
   public abstract void setId(String id);
```

```
    public void ejbLoad() {
    }

    public void ejbStore() {
    }

    public void ejbActivate() {
    }

    public void ejbPassivate() {
    }

    public void setEntityContext(EntityContext entityContext) {
      this.entityContext = entityContext;
    }

    public void unsetEntityContext() {
      this.entityContext = null;
    }

  // metodi di business
    public void addBooks(Collection books) {
      Collection myBooks = this.getBooks();
        myBooks.addAll(books);
    }
  …
  }
```

Si noti come la classe sia contemporaneamente classe astratta ma implementi la EntityBean. Definire l'entity ed i suoi metodi astratti serve per rafforzare il fatto che il bean non si possa considerare completo fino al momento del deploy.

Come noto il metodo ejbPostCreate() entra in funzione dopo la creazione del bean ma prima che questo sia disponibile ai client: fra le varie cose questo può essere utile per interagire con i campi relazionali, dato che un istante prima della chiamata del metodo, non sono utilizzabili.

L'Abstract Programming Model

Tutti i metodi presenti nel bean che si desidera esporre ai client remoti devono essere presenti nella interfaccia remota: i metodi relativi all'Abstract Programming Model però non devono necessariamente essere inseriti nella interfaccia remota, ed anzi in genere è preferibile il contrario per limitare al minimo necessario l'interfaccia remota (e quindi il client–layer) del meccanismo di persistenza astratto.

Inoltre la specifica proibisce che siano esposti metodi relativi ai campi relazionali: come si avrà modo di vedere più avanti, questa regola alquanto saggia, evita che un client vada a manipolare direttamente le relazioni fra bean, limitando l'insorgere di situazioni non coerenti con lo schema relazionale del dominio dei dati.

Infine, come risulta ovvio, l'interfaccia remota non può esporre metodi di modifica della chiave primaria dell'entity.

Relativamente alla interfaccia home esistono importanti differenze rispetto alla precedente specifica: il metodo di ricerca per chiave primaria (findByPrimaryKey()) non è presente, dato che verrà implementato al momento del deploy, mentre gli altri metodi di ricerca verranno definiti tramite la definizione di EJBQL inserite nel file XML di deploy.

Deployment descriptor

Nel deployment descriptor sono state introdotte alcune importanti novità, prima fra tutte il tag <cmp-version> che serve per specificare la versione del CMP utilizzata.
Inserendo

```
<cmp-version>2.x<cmp-version>
```

si specifica che si sta utilizzando il CMP 2.0, mentre 1.x è il valore da utilizzare per la versione precedente. Il primo è il valore di default per cui tale tag non è obbligatorio se si usa CMP 2.0. Il resto del file XML è praticamente identico a prima.

Persistent fields e dependent classes

I campi di persistenza possono essere di tipo primitivo (int, float, …), tipi Java wrapper di primitivi (Byte, Boolean, Short, Integer, Long, Double e Float) oltre al tipo String e Date. Inoltre si possono utilizzare anche tipi Java serializzabili definiti dall'utente: in tal caso questi campi non vengono mappati con i tipi del database ma sono salvati un qualche formato binario.

È fortemente sconsigliato utilizzare questo genere di oggetti, detti dependent classes, a meno che non si debbano memorizzare dati particolari come immagini o dati multimediali; molto probabilmente questi campi possono essere resi persistenti con i tipi standard oppure con un legame relazionale con un altro entity bean. Per i tipi custom serializzabili vale la regola base della serializzazione Java: gli oggetti non si spostano ma verranno passati al client per copia (e non per reference remota); per poter modificare tali attributi si dovrà accedere ai metodi setXXX direttamente sull'entity bean. Relativamente a questi aspetti una buona organizzazione della applicazione dovrebbe sempre prevedere il flusso dei dati tramite oggetti appositi DTO (Data Transfer Object pattern) e limitare se non impedire del tutto l'accesso ai metodi setXXX e getXXX di un entity da parte del client.

Un buon progetto dovrebbe prevedere a fianco degli entity, session bean che fungano da Façade (pattern Session Façade) rimandando a tali oggetti la logica di accesso e manipolazione dei dati. Per maggiori approfondimenti su tali aspetti si può far riferimento a [EJBDesign].

Legami relazionali fra entity bean e relational field

Gli entity bean sono strutture dati che rappresentano la realtà che ci circonda. Per questo motivo, proprio come accade per le strutture dati di database relazionali, gli entity bean spesso sono legati fra loro con relazioni semplici o multiple.

La gestione delle relazioni prima di CMP 2.0 era un compito non troppo semplice: adesso grazie all'utilizzo dello schema di persistenza astratto (ossia alla definizione in XML dei vari aspetti di un entity) il lavoro del programmatore si è notevolmente semplificato.

Gli elementi base delle relazioni fra entity bean sono i cosiddetti campi relazionali (*relational fields*) e sono strettamente legati ad una altra importante novità introdotta in EJB 2.0, le interfacce locali (vedi oltre).

Per affrontare con chiarezza l'argomento si riconsideri la fig. 8.18 in cui è riportato lo schema di relazione fra i vari entity bean: al suo interno è stata descritta una serie di relazioni che per semplicità ruotano (a parte qualche caso particolare) intorno all'entità studente.

Schematicamente le relazioni che si instaurano sono le seguenti

```
Student  (1)  --------> (1)   Desk
Student  (1)  <-------> (1)  Curriculum
Student  (1)  --------> (n)   Book
Student  (n)  --------> (1)   School
Student  (1) <-------> (n)   TestResult
Student  (n) <-------> (1)   StudentClass
Teacher  (n) <-------> (m)  StudentClass
```

Concettualmente non dovrebbe esserci in questi legami e nel simbolismo utilizzato niente di particolarmente complesso da comprendere: il numero tra parentesi indica la *arietà* della relazione, mentre la freccia la navigazione. Ad esempio il primo caso

```
Student  (1)  --------> (1)   Desk
```

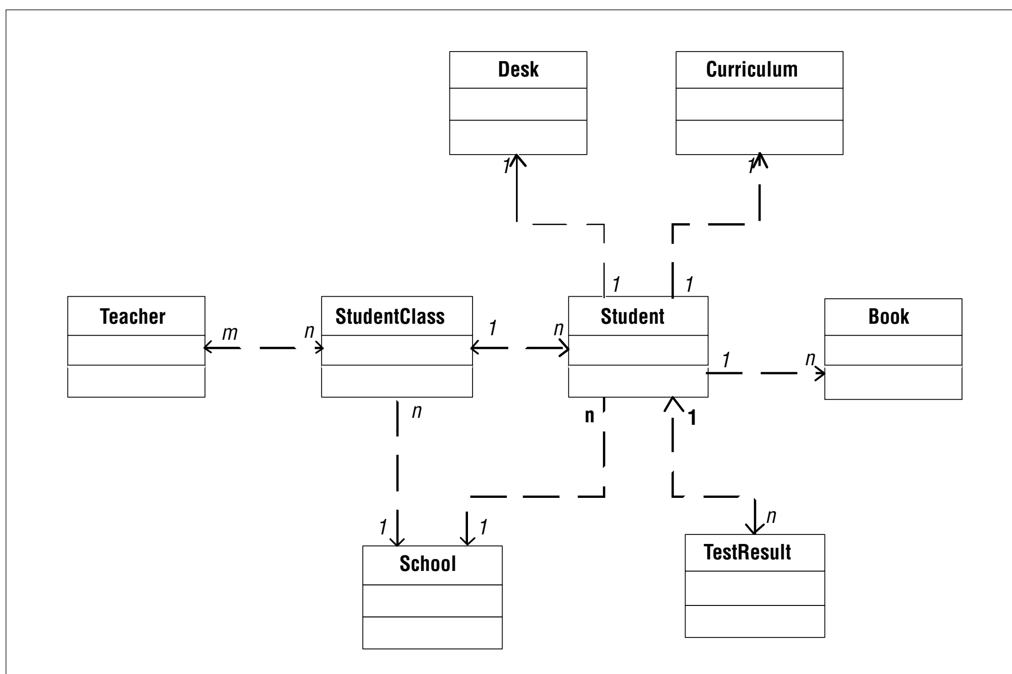
indica che uno studente è legato con un banco, e la navigazione indica che lo studente può ricavare a quale banco è stato assegnato, mentre non è vero il contrario: il banco non conosce nessuna informazione dello studente che vi si siede.

Relational fields: definire il legame

Invece da un punto di vista EJB il comportamento dell'applicazione e dell'application server nel gestire le relazioni, danno luogo ad alcuni aspetti interessantissimi.

Si prenda in esame il caso `Student-Desk`: la relazione viene mantenuta utilizzando un campo di relazione presente nell'oggetto `Student`, mentre in `Desk` non vi è niente che lo leghi con lo Studente (a causa delle considerazioni fatte circa la navigabilità); in `Student` vi sarà quindi un qualche reference a `Desk`, ma non il contrario.

Figura 8.18 – *Organizzazione relazionale degli entity bean della applicazione SchoolManager.*



La specifica impone che i reference necessari per mantenere le relazioni siano istanze della interfaccia locale del bean referenziato.

Più semplicemente in Student deve esserci una variabile di tipo DeskLocal accessibile tramite metodi getXXX e setXXX.

Le variabili locali sono una nuova caratteristica della EJB 2.0 di cui si parlerà ampiamente più avanti: per il momento si può brevemente dire che si tratta di interfacce concettualmente analoghe alle remote (ossia permettono ad un client l'accesso ai metodi di un bean entity o session), ma differiscono da queste ultime perché consentono l'utilizzo di un bean solo all'interno del container e non da remoto. Sono quindi di fatto dei puntatori a bean che referenziano gli oggetti non in modo remoto (nel senso tipico di RMI) ma locale in-JVM (o più precisamente in-container). Detto questo si deve inoltre tenere presente che, essendo in CMP 2.0 astratto il modello di persistenza, la variabile di relazione sarà definita in modo astratto tramite la coppia di metodi

```
public abstract void setDesk(DeskLocal desk);
public abstract DeskLocal getDesk();
```

Se la relazione fosse stata *1* a *n*, come ad esempio per il caso fra studente e libro, il legame non sarebbe rappresentabile con un semplice reference di tipo local, ma con una collection di local:

```
public abstract void setBooks(Collection books);
public abstract Collection getBooks();
```

Nel file ejb-jar.xml di deploy le relazioni sono salvate tramite appositi tag: il tutto inizia con il tag

```
<relationships>
```

mentre il tag ejb-relation specifica il legame (navigabilità e molteplicità) fra i due bean

```
<ejb-relation>
  <ejb-relation-name>student-desk</ejb-relation-name>
  <ejb-relationship-role>
    <description>student</description>
    <ejb-relationship-role-name>StudentRelationshipRole</ejb-relationship-role-name>
    <multiplicity>One</multiplicity>
    <relationship-role-source>
      <description>student</description>
      <ejb-name>Student</ejb-name>
    </relationship-role-source>
    <cmr-field>
      <description>desk</description>
      <cmr-field-name>desk</cmr-field-name>
    </cmr-field>
  </ejb-relationship-role>
  <ejb-relationship-role>
    <description>desk</description>
    <ejb-relationship-role-name>DeskRelationshipRole</ejb-relationship-role-name>
    <multiplicity>One</multiplicity>
    <relationship-role-source>
      <description>desk</description>
      <ejb-name>Desk</ejb-name>
    </relationship-role-source>
  </ejb-relationship-role>
</ejb-relation>
```

La definizione del legame fra due entity non si ferma qui: nel file di deploy proprietario è necessario specificare per ogni relazione quali campi del database relazionale sottostante debbano essere utilizzati per rendere persistente tale relazione (ossia quali chiavi debbano essere agganciate).

Ad esempio se si usa JBoss come application server, all'interno del file jbosscmp-jdbc.xml si troverebbe una sezione dedicata alla definizione della relazione fra i due bean:

```
  <ejb-relation>
```

```
    <ejb-relation-name>student-desk</ejb-relation-name>
    <relation-table-mapping>
      <table-name>student_desk</table-name>
    </relation-table-mapping>
    <ejb-relationship-role>
      <ejb-relationship-role-name>StudentRelationshipRole</ejb-relationship-role-name>
      <key-fields>
        <key-field>
          <field-name>id</field-name>
          <column-name>student_id</column-name>
        </key-field>
      </key-fields>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <ejb-relationship-role-name>DeskRelationshipRole</ejb-relationship-role-name>
      <key-fields>
        <key-field>
          <field-name>id</field-name>
          <column-name>desk_id</column-name>
        </key-field>
      </key-fields>
    </ejb-relationship-role>
  </ejb-relation>
```

Creare il legame

Il legame fra due bean si instaura in modo molto semplice, assegnando il reference alla local del bean Desk nella variabile di relazione di Student.
Ad esempio in Student si potrà trovare il metodo

```
public void assignDeskToStudent(DeskLocal deskLocal) {
  this.setDesk(deskLocal) ;
}
```

Il metodo assignDeskToStudent è esposto nella interfaccia remota del bean Student: questa soluzione permette ad un client remoto ad esempio di impostare il legame di relazione, dato che i campi relazionali non possono essere esposti come remoti.

Le cose sono simili nel caso in cui la relazione sia di tipo *1* a *n*. Si consideri il legame ad esempio fra Student e Book: il metodo di assegnazione dovrà aggiungere il reference local di Book alla collezione di reference di Student

```
public void addBook(BookLocal BookLocal) {
  // ricava la collezione dei libri assegnati a questo studente
```

```
    Collection myBooks = this.getBooks();
    myBooks.add(BookLocal);
  }
```

Anche in questo caso non si espone direttamente la collezione Book, ma lo si fa tramite metodo. La differenza nel nome (assignXXX al posto di addXXX) non segue nessuna specifica particolare ma è solo una convenzione per rafforzare la differenza fra relazione *1* a *1* e *1* a *n*.

Nel caso in cui la relazione fra due oggetti sia più forte, ossia bean A non può esistere senza che sia assegnata una relazione con bean B, allora le cose sono leggermente differenti. È il caso ad esempio del legame StudentClass-School: una classe infatti non può esistere senza che le sia assegnata la scuola di appartenenza. In questo caso il legame fra i due oggetti deve avvenire all'interno del costruttore, più precisamente dentro la ejbPostCreate(). Ad esempio all'interno ejbCreate() di StudentClass si ricevono tutti i parametri (tramite DTO) per la istanziazione degli attributi del bean

```
public String ejbCreate(StudentClassDTO dto) throws CreateException {
  setId(dto.getId());
  setCapacity(dto.getCapacity());
  setDescription(dto.getDescription());
  setLocation(dto.getLocation());
  setName(dto.getName());
  return "";
  }
```

a questo punto nel metodo ejbPostCreate si procede alla assegnazione della relazione con School

```
public void ejbPostCreate(StudentClassDTO dto) throws CreateException {
  try {
    Context context = new InitialContext();
    Object school = context.lookup("School");
    SchoolLocalHome schoolLocalHome
      = (SchoolLocalHome) PortableRemoteObject.narrow(school, SchoolLocalHome.class);
    // ricerca la scuola con l'id e l'assegna al reference locale
    SchoolLocal schoolLocal = schoolLocalHome.findByPrimaryKey(dto.getSchoolDTO().getId());
  }
  catch (Exception ex) {
    … fai qualcosa con l'eccezione
  }
}
```

Si noti come prima si procede alla ricerca della scuola corrispondente all'id passato nel DTO e poi con il reference locale si procede alla assegnazione.

Il motivo per cui questa operazione viene effettuata nella ejbPostCreate() è perché all'interno della ejbCreate() il bean ancora non esiste per cui non sarebbe possibile effettuare l'assegnazione.

Reverse pointers e assegnazioni automatiche

Al momento della assegnazione il container effettua alcune interessanti operazioni a seconda del tipo di relazione che si instaura fra i due bean. Di seguito sono riportati alcuni casi interessanti, per maggiori approfondimenti si può fare riferimento a [EJB3].

Caso 1: relazione *1* a *1* monodirezionale

È il caso appena visto fra Student e Desk: l'assegnazione avviene solamente all'interno di uno dei due bean (quello da cui si può navigare), ad esempio si immagini il caso in cui myStudent1 riceva l'assegnazione di myDesk1. Risulta chiaro che, essendo la relazione 1 a 1, se a myStudent1 viene assegnato myDesk2, allora il bean myDesk1 rimarrà non assegnato a nessun studente. È vero anche il viceversa: assegnando myDesk1 a myStudent2, allora il container rimuove la relazione con myStudent1 che rimarrà senza un banco assegnato, dato che un banco non può essere assegnato a più di uno studente. Si può convenzionalmente definire questo comportamento come *spostamento di reference* e come si avrà modo di vedere nei casi successivi, vale anche per le collection.

Caso 2: relazione *1* a *1* bidirezionale

È il caso di Student–Curriculum. Le considerazioni fatte per il caso precedente sono del tutto analoghe, con una importante differenza. Al momento della assegnazione del reference local di Curriculum nel bean Student (in ejbPostCreate() di Student) non è necessario procedere anche alla assegnazione del reference locale di Student nel bean Curriculum: il container effettua questa assegnazione in automatico. Questa operazione si dice *automatic reverse pointer* e viene gestita dal container che provvede anche alla sincronizzazione delle chiavi nelle tabelle Student e Curriculum (per maggiori dettagli si veda l'appendice dedicata all'esempio).

Caso 3: relazione *1* a *n* monodirezionale

È il caso della relazione fra Student e Book: le considerazioni fatte per il caso 1 sono valide anche qui. La differenza è che uno studente può avere più libri per cui è consentito effettuare assegnazioni multiple di libri allo stesso bean Student. Se invece myBook1 viene aggiunto alla collection di libri di myStudent2, verrà automaticamente rimosso dalla collection di myStudent1. Lo spostamento di reference può essere effettuato anche su intere collezioni di libri, prendendo ad esempio la collection di myStudent1 e assegnandola a myStudent2.

Nel caso in cui il database non sia stato già creato ma si lascia al container la possibilità di creare e gestire i dati come preferisce, è interessante notare come l'application server, invece di creare una tabella di cross per legare la chiave di uno studente con quelle degli *n* libri, preferisca esportare la chiave dello studente nella tabella dei libri.

Caso 4: relazione *n* a *1* monodirezionale

La relazione è presente nell'esempio fra i bean Student e School: la cosa può apparire non molto realistica ma in questo caso si ipotizza che lo studente sia a conoscenza delle informazioni della scuola a cui si è iscritto ma non il viceversa. Si immagini che la anagrafica scuola sia una entità separata. Anche in questo caso lo myStudent1 può essere assegnato ad una sola istanza di mySchool1 mentre alla collection di relazione di School potranno essere aggiunti tutti gli studenti del caso.

Caso 5: relazione *1* a *n* bidirezionale

La relazione è presente nell'esempio fra i bean StudentClass e Student. Questo tipo di relazione è equivalente a quella simmetrica n a 1 bidirezionale per cui si analizza solo questo caso. A parte le considerazioni relative allo spostamento dei reference (implicito per il bean Student che può appartenere solo ad una classe) non vi sono altre particolari implicazioni. Nel soggetto che partecipa con molteplicità 1 vi sarà un reference local all'altro soggetto che invece a sua volta conterrà un reference di tipo collection.

Caso 6: relazione *n* a *m* monodirezionale

Questa relazione non è implementata nell'esempio, ma la sua aggiunta potrebbe essere fatta dal lettore senza particolari elementi a cui porre attenzione, se non quelli elencati nei punti precedenti.

Caso 7: relazione *n* a *m* bidirezionale

È rappresentato dalla relazione che si instaura fra StudentClass e Teacher: in questo caso il legame deve essere mantenuto tramite una tabelle di cross (in questo caso la STUDENT_CLASS_TEACHER, che contiene le associazioni fra le chiavi della tabella STUDENT_CLASS e della tabella TEACHER.

Questa impostazione non si riflette in alcun modo nella applicazione, dato che le associazioni sono mantenute, come al solito, tramite collection nei due bean, e la traduzione in link nella cross table viene fatta dal container.

EJB Query Language: EJBQL

Gli entity di tipo CMP permettono di astrarre completamente il processo di sincronizzazione dei dati con il sistema sottostante grazie ad una raffinata tecnica di definizione del mapping object relational. La specifica 2.0 introduce un altro potente strumento che in tal senso permette di rendere ancor più standard il meccanismo di definizione del comportamento di un CMP. Già da tempo era disponibile un sistema basato su file di script che consentiva di definire in modo astratto il comportamento dei metodi relativi al ciclo di vita. Ad esempio tramite sintassi di vario tipo, era possibile definire il comportamento dei vari metodi di ricerca all'interno del database relazionale o a oggetti.

La grossa limitazione era che tali script, pur ispirandosi in qualche modo all'SQL, erano nella maggior parte dei casi basati su una grammatica proprietaria dei vari application server, limitando fortemente la portabilità della applicazione finita; non era infatti possibile effettuare un nuovo deploy del file .jar contenente uno o più entity CMP da un container ad un altro senza prima ricontrollare che il container fosse in grado di comprendere senza problemi il linguaggio SQL-like. Sun per questo motivo ha definito da specifica un linguaggio simil SQL, detto EJB-QL: esso definisce come il persistence manager debba implementare i metodi di ricerca definiti nella interfaccia home del bean. Basato su SQL-92 esso può essere compilato automaticamente rendendo il deploy del bean e della applicazione complessiva molto più portabile. I vari statement EJBQL sono definiti nel deployment descriptor del bean, tramite una ben precisa sintassi XML. Riconsiderando l'entity Student, potrebbero essere definiti i seguenti metodi di ricerca nella home interface:

```
public StudentRemote findByPrimaryKey(String id) throws FinderException, RemoteException;
public Collection findAll() throws FinderException, RemoteException;
public Collection findByFirstName(String firstName) throws FinderException, RemoteException;
public Collection findByTeacherId(String teacherId) throws FinderException,  RemoteException;
```

Il primo metodo permette di cercare uno studente in base al suo id, mentre findAll() e findByFirstName() effettuano ricerche più ampie utilizzando i parametri in input, e di fatti restituiscono dati aggregati in collection.

Dal momento che la chiave primaria è un dato ben specificato al momento della creazione del bean, non è necessario insegnare al container come realizzare tale ricerca: si tratta di una banale select sulla tabella relativa al bean con filtro sulla chiave della tabella.

Le altre ricerche invece non sono determinabili in modo automatico, per cui si deve procedere alla definizione del codice EJBQL necessario. Il metodo findAll() (che corrisponde ad una select senza filtri) viene così definita in EJBQL

```
SELECT OBJECT(s) FROM Student s
```

Lo statement OBJECT(s) serve per definire un oggetto (o collezione di oggetti) che verranno individuati dal resto della istruzione EJBQL.

Il metodo findByFirstName() invece è così definito in EJBQL

```
SELECT Object(s)
FROM Student as s
WHERE (LOCATE(?1, s.firstName) >0)
```

In questo caso la clausola WHERE permette di identificare se la parola passata come parametro di ricerca (individuate da ?1) è contenuta nella variabile firstName. Si noti l'operatore punto (".") che permette di accedere agli attributi dell'oggetto s, definito di tipo Student tramite la clausola FROM.

Infine il terzo caso mostra due interessanti caratteristiche di EJBQL: l'aggregazione e la navigazione. Il metodo consente di ricercare tutti gli studenti il cui insegnante ha un nome specifico. Ecco la definizione EJBQL:

```
SELECT OBJECT(s)
FROM Student AS s,
IN (s.studentClass.teachers) AS t
WHERE t.id=?1
```

La clausola IN – AS in questo caso compie una operazione simile alla JOIN dell'SQL: permette di associare alla variabile t un set di dati corrispondente agli insegnanti in relazione con lo studente in esame. Dato che la variabile di relazione teachers è una collection contenente le interfacce local degli insegnanti assegnati allo studente non sarebbe stato possibile scrivere s.studentClass.teachers.firstName.

Implicitamente questa istruzione mostra un'altra interessante funzionalità di EJBQL: la navigazione fra bean. Scrivere s.studentClass.teacher permette di muoversi da uno studente verso la classe associata ed infine verso la collection degli insegnanti. Se si pensa al significato di tali legami (aggregati relazionali *1* a *n* o *m* a *n*) si può facilmente comprendere la potenza di questo costrutto e la semplicità con la quale consente di risolvere la navigazione.

I campi di persistenza, dato che non identificano nessuna relazione e quindi non permettono di *vedere* cosa ci sia dall'altra parte, vengono detti a volte anche campi *opachi.*

Ecco un pezzo completo di EJBQL che specifica i metodi di ricerca appena visti:

```
<query>
  <query-method>
    <method-name>findAll</method-name>
    <method-params />
  </query-method>
  <ejb-ql>
    SELECT OBJECT(s) FROM Student s
  </ejb-ql>
</query>
<query>
  <query-method>
    <method-name>findByFirstName</method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
    </method-params>
  </query-method>
  <ejb-ql>
    SELECT Object(s) FROM Student as s WHERE (LOCATE(?1, s.firstName) &gt;0)
  </ejb-ql>
</query>
<query>
  <query-method>
    <method-name>findByTeacherId</method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
    </method-params>
  </query-method>
  <ejb-ql>
    SELECT OBJECT(s) FROM Student AS s, IN (s.studentClass.teachers) AS t WHERE t.id=?1
  </ejb-ql>
</query>
```

Per maggiori approfondimenti ed esempi su EJBQL si può consultare direttamente il tutorial Sun su tale argomento ([EJBQL-Tutorial]).

EJB 2.0: la API Client Local

EJB 2.0 introduce il concetto di *local component interface* che offre una diversa semantica di utilizzo dei componenti enterprise così come un differente contesto di esecuzione, con lo scopo di ottimizzare l'interazione fra bean; questa nuova modalità di invocazione è valida solo nel caso di bean co-located, ossia in esecuzione all'interno dello stesso container e della stessa JVM.

In tale scenario entity e session possono comunicare fra loro senza utilizzare nessuna risorsa di rete, ma basandosi semplicemente sulla semantica di Java RMI-IIOP.

In realtà questo meccanismo non è del tutto nuovo dato che i vari application server, già al tempo della specifica EJB 1.1, effettuavano alcune operazioni in modo trasparente al fine di ottimizzare le prestazioni: nel caso di bean co-located infatti, oltre ad eliminare lo strato della rete per le invocazioni dei metodi remoti, molti realizzavano il passaggio dei parametri per reference, piuttosto che per copia.

Questo modo di operare permette in determinati casi un notevole incremento delle prestazioni, anche se in alcuni casi viola le specifiche o comunque rende meno chiaro lo scenario, dato che non è possibile a priori avere la perfetta conoscenza del comportamento di un application server rispetto ad un altro.

Per questo motivo la nuova specifica impone l'introduzione delle interfacce locali, vietando il passaggio per reference. Questa restrizione non è stata gradita da tutti, ed in molti casi è stata considerata una inutile complicazione della tecnologia EJB: i detrattori del costrutto local infatti asseriscono che sarebbe stato meglio procedere alla sua introduzione ufficiale ma in modo trasparente agli occhi del programmatore EJB (ossia con ottimizzazioni interne agli application server), senza la necessità di introdurre un nuovo modello.

Sebbene questa considerazione si basi su motivazioni valide, è comunque vero che il costrutto local spinge ad una maggiore chiarezza e pulizia in fase di design della applicazione.

Di seguito è riportata la corrispondente versione "local" dell'interfaccia remota del bean Student vista in precedenza:

```
public interface StudentLocal extends EJBLocalObject {
  public void addBooks(Collection books);
  public void assignCurriculum(String curriculumId);
  public void addTestResults(Collection testResults);
  public void addTestResult(TestResultLocal testResultLocal);
  public void addBook(BookLocal BookLocal);
  public void setAddress(String address);
  public String getAddress();
  public void setAge(String age);
  public String getAge();
  public void setBirthday(String birthday);
  public String getBirthday();
  public void setEmail(String email);
  public String getEmail();
  public void setFirstName(String firstName);
```

```
    public String getFirstName();
    public void setLastName(String lastName);
    public String getLastName();
    public void setPassword(String password);
    public String getPassword();
    public String getId();
    public void setCurriculum(CurriculumLocal curriculum);
    public CurriculumLocal getCurriculum();
    public void setBooks(Collection books);
    public Collection getBooks();
    public void setTestResult(Collection testResult);
    public Collection getTestResult();
    public void setStudentClass(StudentClassLocal studentClass);
    public StudentClassLocal getStudentClass();
    public void setSchool(SchoolLocal school);
    public SchoolLocal getSchool();
    public void setDesk(DeskLocal desk);
    public DeskLocal getDesk();
}
```

L'interfaccia local vs. remote

Analogamente alla remote, l'interfaccia local è pensata per esporre i metodi di business di un bean, con l'importante differenza che in questo caso i metodi esposti potranno essere invocati solamente da un altro bean co-located e non da un client remoto.

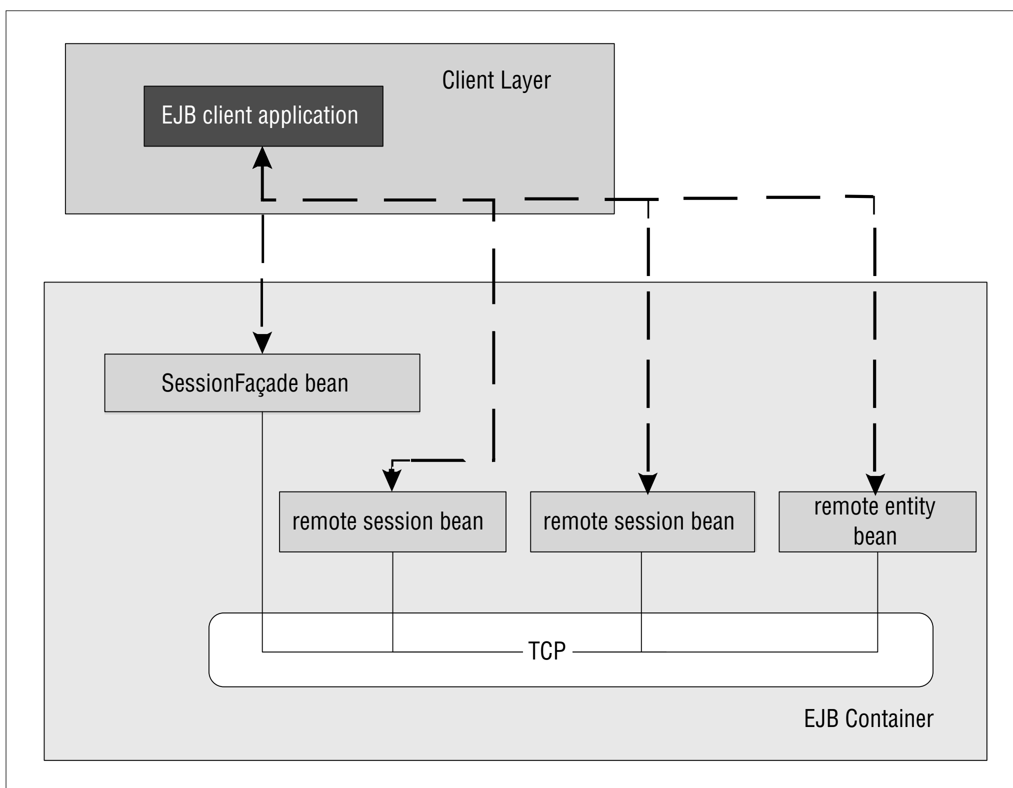
Rimangono del tutto valide le regole di corrispondenza sintattica fra le firme dei metodi del bean e quelli della interfaccia locale, così come accade fra bean e interfaccia remota.

Da un punto di vista progettuale le interfacce locali spingono ad una organizzazione dei componenti più pulita e pattern oriented: fornire un entity bean solamente di interfaccia locale significa proibirne l'accesso da parte di un client remoto, e quindi spinge all'uso ad esempio di un session bean (con interfaccia remota) che svolga il compito di Session Façade nei confronti del client.

I benefici che si possono trarre dall'uso del pattern Session Façade sono molteplici: si migliora la gestione del workflow applicativo, dato che si separa lo strato dei dati da quello di business logic e lo si isola in modo che il client non possa accedervi direttamente (un attributo di uno studente non dovrebbe mai essere modificato direttamente da un client ma solo tramite operazioni di business logic ad opera del session bean).

Ma con il Session Façade si migliora anche l'aspetto prestazionale: con una sola invocazione di un metodo di session si possono eseguire più operazioni automaticamente; ad esempio si potrebbe effettuare una ricerca di tutti gli studenti assegnati ad un determinato insegnante e procedere alla assegnazione di un titolo di libro per codesti studenti. A fronte di una invocazione remota (lenta perché passa per lo strato di rete) si potrebbero avere due o più invocazioni locali (veloci perché in-VM).

Figura 8.19 – Architettura distribuita basata su Session Façade: in questo caso il session bean, benché in esecuzione all'interno della stessa JVM e dello stesso container degli altri bean, deve utilizzare la rete per poterne invocare i metodi. Si noti che, pur contravvenendo al pattern Façade, il client potrebbe accedere anche agli altri bean, essendo definiti come remoti.



L'interfaccia local home vs. home

Analogamente alla local, la local home ha lo scopo di definire quei metodi relativi al ciclo di vita di un bean invocabili da altri bean co-located. Una interfaccia local home estende direttamente dalla EJBLocalHome i cui metodi in questo caso sono quelli di ricerca, i creazionali e quelli di rimozione. Di seguito è riportata l'interfaccia local home dell'entity bean Student:

```
public interface StudentLocalHome extends EJBLocalHome {
  public StudentLocal create(StudentDTO dto) throws CreateException;
  public StudentLocal findByPrimaryKey(String id) throws FinderException;
  public Collection findAll() throws FinderException;
  public Collection findByFirstName(String firstName) throws FinderException;
```

```
    public Collection findByTeacherId(String teacherId) throws FinderException;
}
```

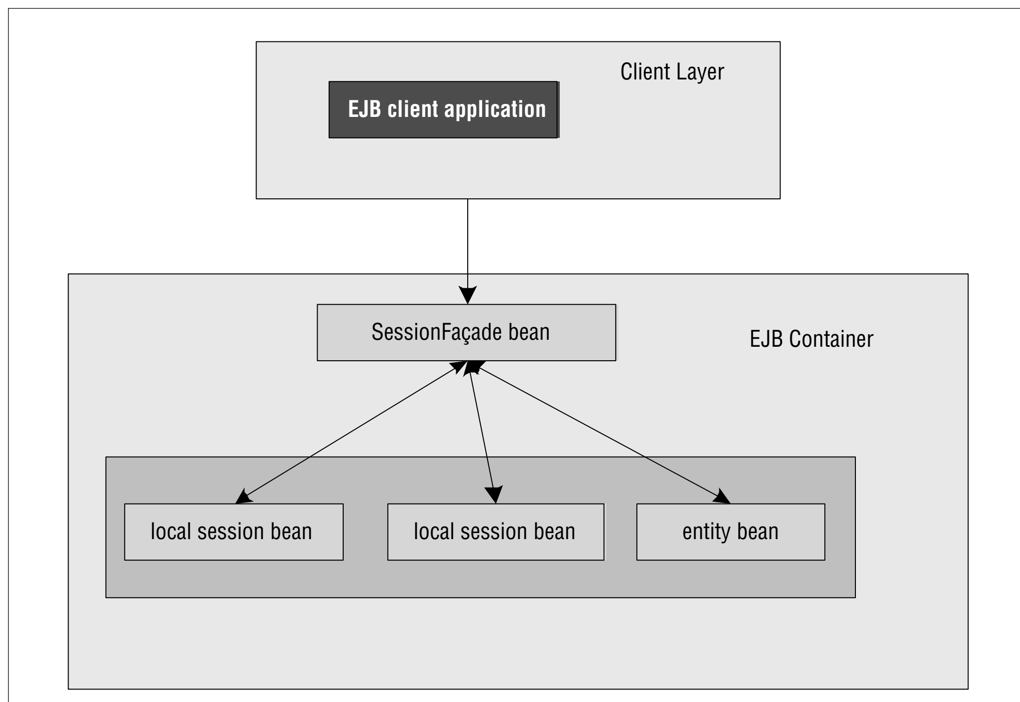
Anche in questo caso valgono le medesime considerazioni circa le firme dei metodi e sulle eccezioni remote; da notare inoltre come in questo caso i metodi creazionali e quelli di ricerca non restituiscono oggetti di tipo remoto, ma istanze discendenti dalla interfaccia locale.

Deployment descriptor

Ovviamente anche il deployment descriptor XML subisce delle modifiche seppure molto semplici: nel caso del session bean Student di cui sopra, il file di deploy ejb-jar.xml deve contenere la definizione delle interfacce locali del session:

```
<session>
```

Figura 8.20 – *Architettura distribuita basata su Session Façade ma con interfacce locali: in questo caso il session façade bean può accedere agli altri bean direttamente con chiamate in-memory, senza dover scendere allo strato di comunicazione RMI-IIOP. Si noti come il client in questo caso non accedere anche agli altri bean, visto che sono definiti come locali.*



```
    <display-name>SchoolManager</display-name>
    <ejb-name>SchoolManager</ejb-name>
    <home>com.mokabyte.mokabook2.ejb.schoolmanager.SchoolManagerHome</home>
    <remote>com.mokabyte.mokabook2.ejb.schoolmanager.SchoolManagerRemote</remote>
    <local-home>com.mokabyte.mokabook2.ejb.schoolmanager.SchoolManagerLocalHome</local-home>
    <local>com.mokabyte.mokabook2.ejb.schoolmanager.SchoolManagerLocal</local>
    <ejb-class>com.mokabyte.mokabook2.ejb.schoolmanager.SchoolManagerBean</ejb-class>
</session>
```

Il punto di vista del client

Per quanto detto fino ad ora dovrebbe essere chiaro che il client di un bean local deve essere per forza un altro bean co-located, in questo caso il session bean façade della applicazione. Come accennato in precedenza la semantica di interazione fra i due bean è del tutto analoga al caso remoto.

Il meccanismo si basa sempre su Java RMI-IIOP e quindi sarà necessario ricorrere alle solite operazioni di lookup della interfaccia home (locale in questo caso) ed alla successiva invocazione dei metodi di creazione (find o create). Ecco un esempio: il metodo assignStudentToStudentClass assegna uno studente (identificato per ID) ad una classe (anch'essa identificata per ID):

```
public void assignStudentToStudentClass(String studentId, String studentCassId) {
  try {

    // si cerca la classe da assegnare allo studente
    Object studentClass = context.lookup("StudentClass");
    StudentClassLocalHome studentClassLocalHome;
    studentClassLocalHome = (StudentClassLocalHome) PortableRemoteObject.narrow(studentClass,
                                                               StudentClassLocalHome.class);
    // potrebbe dare errore se non si trova il record nel db
    StudentClassLocal studentClassLocal = studentClassLocalHome.findByPrimaryKey(studentCassId);

    StudentLocal studentLocal = null;
    Object student = context.lookup("Student");
    StudentLocalHome studentHome;
    studentHome = (StudentLocalHome) PortableRemoteObject.narrow(student, StudentLocalHome.class);
    // potrebbe dare errore se non si trova il record nel db
    studentLocal = studentHome.findByPrimaryKey(studentId);
    studentLocal.setStudentClass(studentClassLocal);
  }
    catch (Exception e) {
      // fai qualcosa con l'eccezione
    }

  }
```

Considerazioni

In EJB 2.0 un enterprise bean può implementare sia le interfacce locali sia quelle remote, dando maggiore flessibilità alla applicazione: a seconda del caso il bean potrà essere quindi utilizzato da un client remoto o da un bean co-located.

Quando scegliere l'una o l'altra soluzione dipende molto dalla architettura scelta e dai pattern architetturali implementati (al solito in [EJBDesign] si trovano ottimi consigli ed indicazioni).

Sicuramente utilizzare interfacce locali aumenta considerevolmente le prestazioni in caso di lookup RMI o di un elevato carico di lavoro. La controindicazione derivante dal loro utilizzo risiede nella minore portabilità delle applicazioni basate su questo costrutto: i bean infatti in questo caso non potranno essere spostati da un container ad un altro o da una JVM ad un'altra e questo potrebbe rappresentare una grossa limitazione nel caso di sistemi cluster o in architetture fault tolerant. Con la Local API quindi si riduce notevolmente il concetto di *location transparency*, rimanendo legati alla piattaforma della invocazione iniziale del bean.

EJB 2.0: i Message Driven Bean

Da quando la specifica EJB è stata presentata al grande pubblico, ha riscosso un grosso successo grazie alle sue caratteristiche che in ottica enterprise ne fanno lo strumento preferito per lo sviluppo di applicazioni distribuite, transazionali, sicure.

Il concetto fondamentale di questa piattaforma è rendere disponibili oggetti remoti i cui metodi possano essere invocati in modo sincrono da client remoti, sotto una stretta policy di sicurezza e controllo da parte del container.

Con il dilagare delle reti geografiche e l'allargarsi del concetto di distribuito, si è fatto sempre più pressante il bisogno di eseguire operazioni in modo asincrono sotto il vincolo di operare in un contesto sicuro e transazionale.

La modalità asincrona si traduce in pratica nella possibilità di scambiare messaggi in grado poi di scatenare altre operazioni o eventi.

Java già da tempo forniva con la JMS API, la possibilità di interagire con middleware di messaggistica distribuiti (i cosiddetti MOM), anche se ad oggi mancava la possibilità di gestire in modo automatico lo scambio di messaggi in chiave J2EE (o più precisamente all'interno di EJB). Anche se tale collegamento poteva essere realizzato in modo più o meno manuale, mancava la possibilità di poter disporre di un bean (session o entity) in grado di restare dormiente fino alla ricezione di un messaggio e di svegliarsi automaticamente per effettuare determinate operazioni.

Con la specifica 2.0 questa carenza è stata colmata. Adesso infatti, grazie alla presenza dei cosiddetti Message Driven Bean (MDB) è possibile mettere in comunicazione tutto il mondo J2EE transazionale. In questo paragrafo si affonderanno i MDB senza entrare nel dettaglio della JMS API per la quale si rimanda al capitolo dedicato in questo libro.

I Message Driven Bean

Un MDB è un bean in cui il concetto di interfaccia remota è del tutto assente, e il solo meccanismo di interazione con il mondo esterno è quello basato sulla ricezione di messaggi

asincroni. Si tratta quindi di componenti stateless transaction aware, il cui unico compito è quello di processare messaggi JMS provenienti da topic o code.

Per questo motivo un MDB è un enterprise bean che funziona come ascoltatore di una particolare destinazione, *coda* o *topic*, in modo del tutto simile a quello che potrebbe fare un listener JMS scritto ad hoc.

Il valore aggiunto che si ha dall'usare un MDB al posto di una normale applicazione client JMS è che in questo caso l'ascoltatore, essendo un enterprise bean, vive completamente all'interno di un contesto sicuro, transazionale e fault tolerant. Un altro grosso vantaggio è che rispetto ad un semplice client, essi sono in grado di processare i messaggi in modo concorrente: similmente a quanto accade per le servlet, lo sviluppatore si deve concentrare sullo sviluppo di un solo componente o meglio nella logica di elaborazione di un messaggio. Al resto pensa il container che potrà eseguire i vari bean in modo concorrente.

Un MDB può ricevere centinaia di messaggi e processarli anche tutti nello stesso istante, dato che il container eseguirà più istanze dello stesso MDB gestiti in modalità di pool.

In modo del tutto analogo a quanto avviene per i session o gli entity, anche i MDB sono gestiti in tutto il loro ciclo di vita e durante l'esecuzione dei metodi di business dal container. Per questo, sebbene essi siano in grado di processare messaggi JMS, l'effettiva ricezione viene gestita dal container, che poi in modalità di callback inoltra la chiamata al bean.

Utilizzare i MDB

Per chi si avvicina per la prima volta a MDB uno dei dubbi ricorrenti è cercare di comprendere come e quando un MDB debba essere utilizzato.

La modalità corretta di utilizzo di un MDB è anche la più semplice che si possa immaginare. Se in un contesto EJB gli entity rappresentano entità astratte, i session degli oggetti di servizio in cui eseguire metodi di business logic, i MDB sono oggetti il cui solo scopo è quello di operare come intermediari per l'esecuzione della business logic in concomitanza di determinati eventi (ossia all'arrivo di un messaggio). Per questo motivo un MDB dovrebbe implementare solamente la logica di gestione del messaggio e dell'analisi delle sue caratteristiche e dar vita così ad un flusso di operazioni invocando la business logic contenuta nei metodi remoti di un sessione bean: ad esempio un MDB potrebbe estrapolare i campi dall'header del messaggio e successivamente stabilire quale metodo di un session invocare passandogli gli opportuni parametri.

Con una metafora piuttosto elementare, un MDB può essere considerato come il controllore di una squadra di operai: è lui che riceve gli ordini dall'alto, ma non esegue nessuna operazione concreta, se non quella di far partire il lavoro dei suoi subordinati.

I MDB possono avere campi di istanza come i session, campi che sono mantenuti durante il loro ciclo di vita, ma essendo stateless e non potendo essere associati ad un client specifico, devono essere gestiti dal container in modo autonomo secondo la logica del pool.

I MDB dall'interno

Per definire un Message Driven Bean è necessario implementare l'interfaccia MessageDrivenBean e la MessageListener. La prima definisce essenzialmente i metodi di callback sui quali il container

interviene per effettuare alcune delle principali operazioni durante il ciclo di vita del bean. La definizione di tale interfaccia è riportata qui di seguito

```
public abstract interface MessageDrivenBean extends EnterpriseBean {
  void ejbRemove() throws EJBException;
  void setMessageDrivenContext(MessageDrivenContext messageDrivenContext) throws EJBException;
}
```

Il metodo `setMessageDrivenContext()` viene invocato dal container per impostare il contesto all'inizio del ciclo di vita dell'MDB

```
MessageDrivenContext ejbContext;
Context jndiContext;

public void setMessageDrivenContext(MessageDrivenContext mdc) {
ejbContext = mdc;
try {
jndiContext = new InitialContext();
}
catch(NamingException ne) {
throw new EJBException(ne);
}
}
```

Invece il metodo `ejbRemove()` viene eseguito al termine del ciclo di vita del bean e può essere utile proprio per ripulire i campi del bean, oltre a chiudere eventuali connessioni con sistemi sottostanti. Per quanto detto nel paragrafo precedente è bene ricordare che un MDB non dovrebbe effettuare connessioni con altri sottosistemi.

Occorre tenere presente che il metodo potrebbe non essere invocato nel caso in cui si generi una eccezione da qualche parte. In tal caso il bean viene immediatamente rimosso dalla memoria e la transazione annullata.

L'interfaccia `MessageDrivenContext` deriva dalla `EJBContext` alla quale non aggiunge nessun metodo ma anzi ne nasconde quei metodi il cui uso non avrebbe senso nel contesto asincrono.

Si tratta dei metodi come `getEJBHome()` e `getEJBLocalHome()` (non esiste una interfaccia home o local home), o `getCallerPrincipal()` o `isCallerInRole()` (non esiste nessun client invocante per un MDB). Dato che un MDB è asincrono il concetto di `CallerRole` perde di significato dal momento che il contesto di sicurezza per ovvi motivi non può essere propagato dal client JMS al bean. Un MDB inoltre deve implementare l'interfaccia `MessageListener` che fornisce al bean la facoltà di interagire con messaggi di tipo JMS. Ecco la sua definizione

```
public abstract interface MessageListener {
  void onMessage(Message message);
}
```

Come si può notare essa è molto semplice, tanto da far sorgere il dubbio sulla sua effettiva utilità: ci si potrebbe chiedere infatti perché definire una interfaccia apposita per la definizione di un solo metodo. Si sarebbe potuto inserire onMessage() direttamente nella interfaccia MessageDrivenBean rendendo il tutto più compatto e semplice.

In realtà nella primissima specifica di EJB 2.0 la scelta fatta fu proprio questa: successivamente però ci si rese conto che questo vincolava un MDB alla possibilità di interagire solamente con messaggi JMS, mentre in futuro potrebbe essere utile creare listener per messaggi di posta elettronica (SMTP listener), JAXM (Java API for XML Messaging) o ebXML. Questi nuovi sistemi richiederebbero differenti implementazioni del metodo onMessage().

Il metodo onMessage() è il luogo dove inserire la business logic del bean: esso viene invocato in modalità di callback non appena il server riceve un messaggio dal topic o queue su cui il bean si è registrato. Non è possibile prevedere a priori in alcun modo quando ciò avvenga, per cui lo sviluppatore deve considerare tale esecuzione del tutto passiva.

Per l'implementazione della business logic al suo interno è altresì importante anche tener presente la filosofia di base sia di JMS che dei MDB.

Un messaggio infatti deve essere considerato come uno strumento per propagare eventi sulla base dei quali poi verranno effettuare determinate operazioni. In una architettura distribuita JMS può essere considerato come uno dei collanti utilizzabili per tenere insieme una struttura eterogenea (in [MBShop-6] viene mostrato come utilizzare i messaggi per coordinare più web application e applicazioni EJB).

Infine, si deve tenere presente che un MDB è una classe Java e come tale può inviare messaggi JMS. Questo aspetto non è molto interessante in ottica EJB, dato che l'implementazione della logica di spedizione verso topic o code non è dissimile da quella presente in ogni altri client Java che si interfacci con un JMS service.

Deployment Descriptor XML

Come ogni altro bean enterprise, anche i MDB utilizzano per il deploy un meccanismo basato su XML, il quale serve per definire alcune informazioni più o meno fisse (il nome del bean) e altre importanti informazioni di tipo variabile (nome del topic/coda, filtri di selezione ed altro ancora).

Il tag principale è sicuramente <message-driven> il quale, definito all'interno di <enterprise-beans>, definisce il nome del bean (tag <ejb-name>), la sua classe (tag <ejb-class>) e il modello transazionale (tag <transaction-type>). Per ovvi motivi all'interno della coppia <message-driven>…</message-driven> non sono presenti tag relativi alle interfacce (es. <home>… </home> <remote>… </remote>) dato che queste non esistono.

Il tag <message-selector> è particolarmente importante e tipico dei MDB: esso permette di specificare una selezione sulla coda/topic, tramite la quale filtrare quei messaggi che dovranno essere ricevuti dal bean. Il concetto di filtro in JMS è molto utile sia per ottimizzare il traffico di rete (riducendo drasticamente il numero di messaggi da inviare ad un determinato ascoltatore), sia per creare ascoltatori più specifici e quindi migliorare l'architettura complessiva ed il codice applicativo.

Un filtro è una regola definita sul server tramite la quale solo determinati messaggi verranno recapitati al client JMS e quindi al MDB.

La sintassi con cui si definisce un selector è molto simile all'SQL-92: chi fosse interessato ad approfondire questi aspetti potrà trovare alcuni semplici esempi in [EJB3] a cui si rimanda per ulteriori approfondimenti.

Il tag <acknowledge-mode>

Quando un messaggio viene inviato da un client verso una coda o un topic di un JMS provider, quest'ultimo provvede ad inoltrarlo verso tutti i listener registrati su tale destinazione. La consegna del messaggio viene certificata dal ricevente che invia al provider una notifica di ricezione, una specie di ricevuta di ritorno. Nel caso in cui il provider non riceva risposta in un determinato lasso di tempo, il messaggio verrà nuovamente inoltrato fino a quando la notifica non sia inviata.

In ambito EJB è il container che comunica al service JMS che un MDB ha ricevuto e consumato correttamente il messaggio, ossia che il metodo onMessage() è stato eseguito correttamente.

In fase di deploy è possibile scegliere la modalità di notifica della ricezione (immediata o ritardata).

Nel caso in cui vi sia una transazione in ballo, l'impostazione scelta per la notifica della ricezione viene ignorata, dato che la notifica avviene in funzione dell'esecuzione della transazione stessa: se la transazione avviene con successo la notifica viene effettuata automaticamente, in caso contrario non viene inviata nessuna notifica.

Se la transazione è di tipo container managed, come nella maggior parte dei casi, l'impostazione scelta nel deploy viene ignorata dal container che procede secondo le policy interne del server.

Nel caso in cui la transazione sia a carico del bean o NotSupported, viene dunque utilizzato il valore di acknowledge-mode, che può assumere uno dei due valori: Auto-acknowledge e Dupsok-acknowledge.

Nel primo caso la notifica è automatica ed effettuata dal container subito dopo che il messaggio è stato processato, mentre con Dups-ok-acknowledge il container può inoltrare la notifica in un secondo momento, quando si ritiene più opportuno farlo (ossia ad esempio quando non vi sono altre operazioni importanti da effettuare con risposta immediata).

Questa seconda possibilità in genere viene scelta per ridurre il carico di lavoro del container ed anche per ridurre al minimo il traffico di rete: dato però che il server EJB ed il provider JMS comunicano in modo molto stretto e frequente, tale variazione non influisce più di tanto sulle prestazioni complessive. Inoltre siccome il ritardo con cui la notifica viene inviate non è prevedibile, potrebbe accadere che il provider JMS ipotizzi la perdita del messaggio, provvedendo a reinviarlo nuovamente.

In questo caso si dovrà provvedere a livello applicativo all'interno del metodo onMessage() ad evitare di processare nuovamente il messaggio.

Per questi motivi quindi in genere questa modalità di notifica non viene utilizzata spesso.

Specificare le destinazioni: il tag <message-drive-destination>

Per la definizione della coda o topic su cui il bean resterà in ascolto si utilizza il tag <message-drive-destination>, per il quale sono previsti i valori javax.jms.Queue e javax.jms.Topic.

Nel caso in cui si specifichi un topic come destinazione dei messaggi, si deve includere anche il tag <subscription-durability> che può assumere i valori Durable e NonDurable. Il concetto di durable

e non durable in questo caso rispecchia fedelmente quanto valido per un normale sistema JMS non EJB: nel primo caso la sottoscrizione a quel topic è di tipo duraturo, ossia nel caso di una disconnessione temporanea del bean dal topic, alla riconnessione verranno recapitati anche quei messaggi prodotti durante il tempo in cui il bean non era in ascolto. È questo un meccanismo che permette ad un client JMS, e quindi anche ad un MDB, di ricevere tutti i messaggi prodotti, senza nessuna perdita.

I motivi per cui un MDB potrebbe disconnettersi dal topic potrebbero essere molteplici, sia per esplicita richiesta del bean, sia per un problema occorso al server EJB.

Nel caso in cui la sottoscrizione sia NonDurable, tutti i messaggi inviati durante il periodo di disconnessione verranno persi; in questo caso si ha un minor livello di sicurezza, anche se le performance sono maggiori.

Nel caso in cui si utilizzi una coda, per il modello di messaggistica di tipo p2p, il concetto di durabilità di un messaggio non ha importanza.

Infine nel file XML di deploy sono presenti altri elementi di tipo <ejb-ref> e <ejb-local-ref> il cui significato dovrebbe essere ormai noto: essi permettono al bean di accedere alle proprietà del JNDI ENC, in modo molto simile a quanto accade per i sessione e per gli entity.

Ciclo di vita di un Message Driven Bean

Il ciclo di vita di un MDB è sicuramente molto più semplice rispetto a quello degli entity o dei session. Infatti, dato che un MDB non è associato a nessun client ma lavora completamente in modalità passiva su invocazione da parte del container, perdono di significato i concetti di attivazione e di passivazione.

I due stati possibili quindi sono solamente Does Not Exist e Method Ready Pool. Il significato di entrambi è piuttosto intuitivo: nel primo caso il bean non esiste in memoria, mentre nel secondo esso è stato creato ed è pronto per essere eseguito all'interno di un pool di oggetti, la cui dimensione, determinata dal container, varia in funzione del carico di lavoro.

Durante il passaggio dallo stato Does Not Exist a quello di Method Ready Pool vengono effettuate fondamentalmente tre operazioni: per prima cosa viene creata una nuova istanza tramite il metodo Class.newIstance(); successivamente viene invocato dal container il metodo setMessageDrivenContext(), con il quale il bean riceve il contesto di esecuzione: memorizzato in un campo del bean.

Infine viene invocato il metodo ejbCreate() del quale esiste una sola versione senza parametri.

Dato che non esistono attivazione e passivazione, un MDB potrebbe in teoria mantenere aperte le connessioni per tutto il periodo di vita, anche se per quanto detto in precedenza un bean di questo tipo non dovrebbe dover aprire connessioni verso sottosistemi particolari.

Nel momento in cui un bean è nel pool, esso è in grado di processare tutti i messaggi inviati presso un topic o una coda. L'elaborazione simultanea avviene grazie alla presenza di più istanze nel pool, dato che ogni istanza può processare un solo messaggio per volta.

Durante il passaggio inverso da Method Ready Pool a Does Not Exist verranno effettuate le operazioni inverse rispetto al processo opposto. Il processo parte dalla invocazione del metodo ejbRemove() dove dovranno essere eseguite tutte le operazioni di chiusura necessarie. All'interno di ejbRemove() continuano ad essere disponibili il MessageDrivenContext e l'ambiente JNDI ENC.

Di seguito è riportato un esempio completo di un deployment descriptor XML di un MDB BulletinMDB non completamente implementato in SchoolManager ed il cui scopo dovrebbe essere quello di gestire un sistema molto semplice di newsletter per gli studenti della scuola.

```
<message-driven>
  <display-name>BulletinMBD</display-name>
  <ejb-name>BulletinMBD</ejb-name>
  <ejb-class>com.mokabyte.mokabook2.ejb.schoolmanager.BulletinMBDBean</ejb-class>
  <transaction-type>Container</transaction-type>
  <message-driven-destination>
    <destination-type>javax.jms.Queue</destination-type>
  </message-driven-destination>
</message-driven>
```

EJB 2.1: il Timer Service e Timer Bean

All'interno dello scenario J2EE, ed in particolare in EJB è sempre crescente l'interesse nel gestire meccanismi di schedulazione automatica in grado di lavorare direttamente con enterprise bean della applicazione. Nella specifica 2.1 è stato introdotto un supporto seppur limitato per questo tipo di servizio. In realtà già in J2SE era stato introdotta la classe java.util.Timer, che consente la schedulazione temporale di thread paralleli, anche se ovviamente è del tutto insufficiente in ottica J2EE. Il Timer Service è un nuovo servizio aggiunto al container con la specifica 2.1: esso consente di attivare timer per determinati periodi di tempo o allo scadere di determinate date.

Al momento in cui si scrive questo capitolo la specifica non è ancora stata rilasciata in forma definitiva e quindi non è possibile testare quanto qui descritto su un application server particolare. Per questo motivo gli argomenti qui trattati potranno differire dalla versione definitiva e quindi è compito del lettore verificarne la validità quando i vari prodotti commerciali e non, implementeranno tale specifica.

Il TimerService Interface

Questo nuovo servizio inserito all'interno del container consente ad un enterprise bean di essere notificato quando una determinata data scade o al trascorrere di un determinato intervallo di tempo. Per utilizzare il servizio un enterprise bean deve implementare l'interfaccia javax.ejb.TimeObject che definisce un solo metodo di callback ejbTimeout()

```
public interface TimeObject{
  public void ejbTimeout(Timer timer)
}
```

Quando il tempo timer scade oppure una determinata data arriva, il metodo ejbTimeout() viene invocato dal container.

Per convenzione nella specifica si parla di scadenza di un timer: tale definizione intende il momento in cui il timer viene mandato in esecuzione, o per il raggiungimento di una data o per lo scadere di un intervallo di tempo.

Per essere attivato un bean deve ricavare un reference all'oggetto TimerService direttamente dall'EJBContext: questo consente al bean di registrarsi come ascoltatore della scadenza di una data o dello scadere di un determinato periodo.
Il codice seguente mostra come ciò sia possibile:

```
Calendar time = Calendar.getInstance();
time.add(Calendar.DATE, 60);
Date date = time.getTime();
EJBContext context = …
TimerService timer = context.getTimerService();
Timer.createTimer(date, null);
```

Nel codice precedente viene per prima cosa creato un oggetto Calendar, poi lo si incrementa di 60 giorni in modo da rappresentare la scadenza del timer. Il timer vero e proprio viene ottenuto dal context EJB tramite il metodo createTimer() del Timer Service.

Tale interfaccia fornisce ai bean la possibilità di accedere al timer service in modo da creare timers con determinate scadenze fisse o con scadenze ricorrenti.

```
package javax.ejb;
import java.util.Date;
import java.io.Serializable;
public interface TimerService {

  // Crea un single-action timer che scade ad una data specifica
  public Timer createTimer(Date expiration, Serializable info)
    throws IllegalArgumentException,IllegalStateException,EJBException;

  // Crea un single-action timer che scade dopo lo scorrere di un determinato periodo di tempo
  public Timer createTimer(long duration, Serializable info)
    throws IllegalArgumentException,IllegalStateException,EJBException;

  // Crea un interval timer che parte ad una specifica data
  public Timer createTimer(Date initialExpiration, long intervalDuration, Serializable info)
    throws IllegalArgumentException,IllegalStateException,EJBException;

  // Create un interval timer che scade dopo lo scorrere di un determinato periodo di tempo
  public Timer createTimer(long initialDuration, long intervalDuration, Serializable info)
    throws IllegalArgumentException,IllegalStateException,EJBException;

  // Ricava tutti i timer attivi associati al bean
```

```
    public java.util.Collection getTimers()
      throws IllegalStateException,EJBException;

}
```

Ogni differente implementazione del metodo createTimer() consente di creare timer in modo diverso, ed in particolare di creare i due tipi di timer previsti: i single-action timer e gli interval-timer.

I primi corrispondono a timer che scadono in determinati momenti, mentre gli altri scadono in modo continuativo ogni *n* millisecondi.

Un timer al momento della creazione viene reso persistente presso un qualche sistema di memorizzazione secondaria: se il sistema si blocca o va in crash per qualche motivo, al riavvio il timer torna ad essere attivo nuovamente.

Timer Interface

Un timer enterprise bean è un oggetto che implementa l'interfaccia javax.ejb.Timer, cosa che permette di creare session entity e MDB in grado di ricevere notifiche temporali dal Timer Service.

Un timer può essere ricavato tramite l'invocazione del metodo TimerService.getTimer() o TimerService.getTimers().

Il metodo TimerObject.ejbTimeout() è quello invocato in callback dal container ricevendo un oggetto che implementa Timer come parametro direttamente dal container. L'interfaccia Timer ha la seguente definizione:

```
package javax.ejb;

public interface Timer {

    // Cancella il timer
    public void cancel()
      throws IllegalStateException,NoSuchObjectLocalException,EJBException;

    // Restituisce le informazioni associate al timer al momento della creazione
    public java.io.Serializable getInfo()
      throws IllegalStateException,NoSuchObjectLocalException,EJBException;

    // Restituisce la prossima scadenza del timer
    public java.util.Date getNextTimeout()
      throws IllegalStateException,NoSuchObjectLocalException,EJBException;

    // Restituisce in millisecondi la prossima scadenza del timer
    public long getTimeRemaining()
```

```
      throws IllegalStateException,NoSuchObjectLocalException,EJBException;

    // Restituisce in forma serializzata un handle al timer
    public TimerHandle getHandle()
      throws IllegalStateException,NoSuchObjectLocalException,EJBException;
}
```

Una istanza di Timer rappresenta esattamente un evento temporale e può essere utilizzata per annullare il timer, per ricavare un handle serializzato, per ricavare i dati applicativi associati al timer, o per ricavare le informazioni temporali sulla scadenza del timer stesso.

Ad esempio il metodo getInfo() restituisce una versione serializzata del Timer tramite la quale ricavare tutte le informazioni di cui sopra.

Il metodo getHandle() invece restituisce una istanza di TimeHandle, simile alla java.ejb.Handle o java.ejb.HomeHandle: essa potrà essere salvata da qualche parte in forma serializzata e permette di ricavare il Timer in un secondo momento. La definizione di TimerHandle è la seguente:

```
package javax.ejb;
public interface TimerHandle extends java.io.Serializable {
  public Timer getTimer() throws NoSuchObjectLocalException, EJBException;
}
```

Quando si invoca il metodo createTimer() l'operazione viene completata all'interno dello scope della transazione attuale. In caso di una rollback l'operazione viene annullata, ossia il timer non viene creato.

Entity Bean Timer

L'utilizzo di entity bean temporizzati può essere particolarmente utile, anche se di fatto introduce uno strato di business logic all'interno del dominio dei dati, per cui viola in qualche modo alcune delle best practice J2EE, secondo le quali un entity è una rappresentazione OO di dati resi persistenti da qualche parte. Se per un momento ci si dimentica di questa indicazione, l'accoppiata entity-timer offre particolari vantaggi.

Allo scadere del tempo del timer, il container ricava la chiave primaria del bean e procede al caricamento dei dati per istanziare l'entity stesso. Quando esso si trova nello stato di ready viene invocato il metodo ejbTimeout() che effettuerà le operazioni contenute al suo interno. Volendo limitare al minimo la violazione delle indicazioni date dalle best practice, si dovrà limitare l'intervento solamente ai dati propri dell'entity. In quest'ottica l'utilizzo dei timer è particolarmente utile: si tratta di strutture dati che modificano se stesse a determinati intervalli di tempo.

Stateless Session Beans Timers

L'utilizzo di session stateless temporizzati permette di implementare meccanismi automatici e transazionali che eseguono compiti in batch in modo semplice e veloce: ripulitura di dati nel

database, trasferimento di record, monitorazione di sottosistemi particolari. Si tratta fondamentalmente di un agente transazionale che vive all'interno del container EJB, con tutti i benefici che questo ne comporta.

Differentemente dal caso precedente, dove un timer è associato ad un determinato entity, qui il timer è associato ad un particolare tipo di session. Così come l'associazione session stateless-client è generica. Quando il timer scade, una qualsiasi istanza del session viene prelevata dal pool e viene eseguito il metodo `ejbTimeout()`.

Message Driven Bean Timers

Discorso del tutto analogo può essere fatto per i bean di tipo message driven. Allo scadere del tempo, viene prelevato un MDB dal pool di quelli disponibile ed eseguito il metodo `ejbTimeout()`.

Da ricordare che un MDB viene inizializzato al momento della ricezione del primo messaggio: molti esperti e progettisti criticavano la mancanza nella prima release della specifica EJB 2.1 della associazione MDB-Timer Service. Ora che questa lacuna è stata colmata, personalmente resto con il dubbio del reale significato di tale associazione. Un timer, così come un MBD, è un oggetto enterprise che sono attivati in corrispondenza di un evento esterno (il tempo o un messaggio): per questo vedo un po' promiscua la mescolanza fra i due domini di lavoro. Dato che i progettisti hanno sentito la necessità di introdurre tale funzionalità, anche in risposta di molte richieste da parte di importanti esperti del settore, è presumibile che questi miei dubbi non siano giustificati.

Al momento manca un modo standardizzato per inizializzare in fase di deploy i MDB-Timers: di fatto essi sono inizializzati al momento della ricezione del primo messaggio, cosa che obbliga ad esempio ad inviare un messaggio automaticamente allo start della applicazione alla coda o al topic su cui il MDB si è registrato. Ciò rappresenta una grossa limitazione, che di fatto riduce l'utilità di questi oggetti. Se includere nella specifica la possibilità di configurare al deploy è una decisione che spetta all'EJB expert group: per il momento non resta che attendere il rilascio della release finale di EJB 2.1.

Problemi

L'interfaccia ed il meccanismo di creazione dei timer hanno ancora molti aspetti da mettere a punto. Ad esempio, come ricordato in [EJB2.1], è alquanto difficile creare e gestire eventi ricorrenti con la stessa flessibilità offerta da Cron, il tool di amministrazione temporizzata presente su Unix. Sebbene questi argomenti esulano dalle prospettive di questo capitolo, si tratta di argomenti alquanto importanti. Per maggiori approfondimenti si rimanda al link sopraccitato.

Conclusioni

L'introduzione del Timer Service in un container EJB è sicuramente una cosa molto importante. Il poter effettuare operazioni a determinati periodi di tempo permette di includere un altro importante servizio nelle applicazioni EJB, consentendo di eliminare applicazioni esterne

(demoni o agenti) che dovrebbero essere eseguite in modalità standalone fuori dal contesto transazionale e fault tolerant tipico di EJB.

Inoltre questa modifica permette di semplificare considerevolmente il processo di configurazione di tali demoni i quali non necessiteranno di appositi sistemi server side (JVM, file di configurazione e meccanismi manutenzione) essendo il tutto contenuto in un container che unisce i vari servizi server side.

La specifica sui Timer Beans è inserita all'interno della specifica 2.1 della quale si possono avere maggiori informazioni presso [JSR153].

Riferimenti bibliografici

[EJB]
Specifiche EJB 2.0
http://java.sun.com/products/ejb

[JNDI]
Java Naming and Directory Interface API
http://java.sun.com/products/jndi

[EJB2.1]
R. MONSON HAEFEL, *The EJB 2.1 Specification: the Service Timer* , "TheServerSide.com"
http://www.theserverside.com/resources/articles/MonsonHaefel-Column4/article.html

[JSR153]
JSR 153 Enterprise JavaBeans 2.1
http://jcp.org/en/jsr/detail?id=153

[EJB3]
R. MONSON-HAEFEL, *Enterprise Java Beans: sviluppare componenti Enterprise Java*, Hops

[EJBDesign]
F. MARINESCU, *EJB Design Pattern*, Wiley

[EJBQL-Tutorial]
EJBQL Tutorial
http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/EJBQL.html

[JMS]
Specifiche JMS
http://java.sun.com/products/jms

[MBShop-6]
G. PULITI, *MokaShop, il negozio online di MokaByte: realizzare applicazioni J2EE multicanale. VI parte: l'integrazione tramite JMS*, "MokaByte", ottobre 2002, http://www.mokabyte.it/2002/10

Capitolo 9

Web service

MASSIMILIANO BIGATTI

Introduzione

I web service sono per certi versi ancora una materia in fase di definizione. Le tecnologie di base sono in continua evoluzione e specifiche complementari nascono con una certa frequenza. Non è chiaro cosa andrà a formare la definizione precisa dei web service, mentre si ha qualche idea sulle tecnologie che sono già affermate. Sicuramente non esiste una sola definizione di web service, ma quella che riteniamo più precisa è la seguente:



Un web service è un componente riutilizzabile che può essere richiamato attraverso tecnologie Internet e che dialoga in XML.

Questa definizione ha svariate implicazioni e apre a possibili interessanti scenari d'utilizzo. Si noti che la definizione non fa riferimento ad alcuno standard specifico, fatta eccezione per XML e Internet.

Servizi e componenti

Un web service è un componente software che svolge un determinato servizio. Può occuparsi di eseguire una determinata elaborazione oppure di scatenare processi complessi in una rete aziendale. Un esempio di semplice componente è quello che, forniti due numeri, ne restituisce la somma. Un caso più complesso è un servizio bancario per disporre il pagamento di bonifici. Un qualsiasi componente remoto che svolga una qualsivoglia funzione può essere considerato un servizio.

Questo aspetto fa emergere una certa similitudine con i componenti remoti nella classica programmazione distribuita. Tecnologie quali Java RMI, CORBA e DCOM sono standard che

consentono l'implementazione di sistemi a più livelli. Possono essere dunque considerati anche questi come web service? La risposta è no, in quanto queste tecnologie utilizzano una comunicazione di file (il protocollo a basso livello) binario e non XML.

Inoltre, queste tecnologie consentono e incoraggiano una granularità applicativa minore rispetto ai web service. Questi ultimi sono pensati per esporre servizi più a "grana grossa" (*coarse-grained*), mentre le tecnologie di programmazione distribuita di cui sopra possono venir utilizzate per strutturare componenti più a "grana-fine" (*fine-grained*). In definitiva i web service espongono poche e semplici funzioni ad alto livello, i componenti remoti possono esporre chiamate a basso livello legate all'implementazione del programma. In questo senso, l'astrazione dei web service è maggiore.

Internet

I web service dialogano e operano attraverso protocolli utilizzati in Internet. Questa scelta è stata adottata per più motivazioni: per prima cosa, il successo di Internet ha decretato una diffusione enorme dei suoi standard, sia a livello di supporto del software e dell'hardware disponibile in azienda, sia nelle competenze e conoscenze tecniche degli operatori del settore.

Utilizzare i protocolli di Internet vuole dire riutilizzare le infrastrutture esistenti e dunque ridurre le problematiche di integrazione. Sebbene si parli di protocolli Internet in generale, lo standard più affermato per veicolare il dialogo con i web service è il protocollo HTTP. Come si vedrà, le specifiche tecniche del protocollo di comunicazione più affermato, SOAP, prediligono questo standard, sebbene rimangano spazi aperti anche per SMTP e FTP.

Il fatto di utilizzare Internet per la comunicazione, inoltre, apre la connettività potenziale a una rete globale di possibili utilizzatori. Esporre un servizio web su Internet equivale a proporne l'utilizzo a tutti quanti abbiano una connessione di rete: un enorme bacino di utenza per potenziali acquirenti di servizi.

Inoltre i protocolli Internet hanno il vantaggio di passare agevolmente attraverso i firewall. Uno degli elementi che ha sempre frenato l'adozione di tecnologie quali Java RMI e CORBA per la comunicazione tra diverse imprese e in generale in Internet è la necessità di configurazioni particolari di firewall per consentire il passaggio delle comunicazioni basate su questi standard.

Sia il fornitore del servizio che il fruitore avrebbero dovuto modificare la configurazione di questi dispositivi di sicurezza per garantire il passaggio dei dati di comunicazione. Questa necessità si trova però spesso in antitesi con le necessità di sicurezza delle aziende. I web service, utilizzando protocolli già consentiti dai firewall, non hanno questo tipo di problematica.

Infine, è bene notare che classiche tecnologie di programmazione distribuita richiedono elementi infrastrutturali aggiuntivi con i relativi costi di acquisizione e manutenzione. CORBA, ad esempio, richiede un broker, un componente che si occupi di coordinare la comunicazione tra gli oggetti distribuiti.

XML

Per finire, i web service comunicano le loro informazioni in XML, il che porta numerosi vantaggi, ma anche qualche svantaggio.

Per prima cosa, XML ha il vantaggio di essere nato per consentire la comunicazione tra diverse piattaforme. Questo significa che un blocco XML prodotto da un programma Java su Solaris può essere letto da un programma C su Windows. Oppure da un palmare. Oppure da un programma COBOL su un mainframe.

Lo standard XML risolve già, a livello di informazione, tutte quelle problematiche di integrazione con sistemi eterogenei. Non è necessario preoccuparsi di problematiche di encoding diverse, oppure della formattazione delle informazioni. Un blocco XML ha lo stesso contenuto su tutte le diverse piattaforme.

Inoltre, XML ha la particolarità di essere un protocollo testuale (e non binario, come quello utilizzato in RMI o CORBA) quindi facilmente leggibile dall'essere umano. Questo è un vantaggio in fase di debug dell'applicazione in quanto il valore dei singoli campi è immediatamente visibile, ma ha lo svantaggio di richiedere più potenza di elaborazione rispetto a protocolli binari, più vicini alla macchina. Si consideri però che un minimo di astrazione è necessario anche per ottenere l'interoperabilità.

Per finire, come i protocolli Internet, anche XML sta ottenendo un'ottima accettazione da parte dell'industra informatica che si traduce nella disponibilità di strumenti di sviluppo e controllo e nella conoscenza della comunità.

Novità o evoluzione?

I web service sono dunque la riproposizione in chiave moderna — o Internet/XML — della vecchia programmazione distribuita? In gran parte la risposta è sì. Non ci troviamo davanti a tecnologie rivoluzionarie, ma piuttosto a una evoluzione realizzata per ovviare alle problematiche che sono sorte con le classiche soluzioni di programmazione distribuita.

Un'evoluzione che però non è partita, come spesso accade, dalla base già esistente in quanto, a parte i concetti astratti, non è stato mantenuto alcuno standard esistente. La motivazione è da ricercarsi anche in aspetti strategici: da una parte, il mondo proprietario di Microsoft ha sempre desiderato imporre DCOM come protocollo universale di comunicazione, ma purtroppo nella sua visione l'universalità di DCOM si sarebbe dovuta realizzare con la presenza del solo Windows su tutti i computer del mondo (anche sui mainframe?). Dall'altra parte CORBA, sviluppato da un consorzio come standard e non come prodotto, ha avuto una accettazione soprattutto nel mondo Unix, in contrapposizione con DCOM che ha avuto uno sviluppo maggiore sui sistemi Windows. Sebbene siano a disposizione prodotti per l'integrazione dei due mondi, le problematiche infrastrutturali hanno posto sempre un freno all'adozione di queste tecnologie.

Alcuni ritengono, inoltre, che Microsoft fosse alla ricerca di qualche cosa di nuovo per promuovere .NET che, se avesse mantenuto il livello attuale di utilizzo di tecnologie proprietarie (DCOM) non avrebbe riscosso un grande successo con gli sviluppatori. Da qui la nascita di SOAP e dei web service che comunque ha come risultante la possibilità di liberarsi dal vincolo di utilizzare DCOM per parlare con sistemi Microsoft.

Java, dal canto suo, ha sviluppato uno standard di programmazione distribuita proprietaria, invece che rivolgersi ai protocolli esistenti, frammentando ulteriormente il mercato. C'è comunque da riconoscere che RMI si è evoluto nel tempo supportando il protocollo di comunicazione IIOP di CORBA, di fatto rendendo la piattaforma Java un'ulteriore piattaforma abilitata a CORBA.

Si noterà, comunque, come molte tecnologie dei web service riprendano concetti già sviluppati nelle classiche tecnologie per la programmazione distribuita, ad esempio in ambito dei registri di web service.

Registri di web service

Il tipico flusso di lavoro di un web service non è però limitato al solo protocollo di trasporto XML/Internet accennato sopra. La visione completa coinvolge ulteriori componenti: i registri di servizi.

Questi elementi rispondono sostanzialmente alla domanda: "Come fanno i singoli potenziali utilizzatori di web service a sapere quali e quanti servizi sono disponibili e come accedere a questi?".

I registri di servizi non sono quindi altro che una sorta di pagine gialle elettroniche che catalogano, ordinano e rendono disponibili per ricerche gli elenchi di web service disponibili sul mercato. Inoltre, permettono di scaricare la documentazione tecnica dei servizi che consente agli utilizzatori di implementare i client che ne vogliono fare uso.

I registri sono per forza di cose entità a livello mondiale: il web di fatto annulla i limiti delle nazioni e dei continenti consentendo, almeno al livello tecnico, l'interoperabilità tra entità anche enormenente distanti tra di loro.

Vista l'enorme potenziale quantità di informazione — si immagini un elenco dei più disparati servizi, forniti da società disseminate in tutto il mondo — è d'obbligo un meccanismo di ricerca fortemente basato sulla tassonomia, cioè sulla catalogazione precisa delle informazioni.

Ma le informazioni tecniche non sono le uniche presenti nei registri: anche informazioni puramente commerciali, come indirizzo, numero di telefono ed e-mail possono essere pubblicate e successivamente estrapolate al fine di instaurare un contatto commerciale tra due parti.

Flusso di lavoro

La sequenza tipica delle operazioni nel mondo dei web service prevede dunque una serie di passaggi che iniziano dal registro dei servizi e terminano con il protocollo di comunicazione.

In particolare, uno scenario abbastanza completo e orientato all'integrazione B2B è il seguente (in seguito si vedranno esempi completi anche non indirizzati al B2B):

- La società A desidera fornire attraverso Internet un servizio: ricerca dunque nel registro la tipologia di servizio che vuole offrire. Se non esiste una tipologia conforme o se quelle trovate non dovessero soddisfarla, ne può creare una ad hoc.

- Con lo schema che formalizza la definizione del servizio alla mano, il reparto IT prepara una implementazione del servizio.

- Il web service viene pubblicato sul registro così che possa essere trovato dai potenziali clienti.

- A questo punto la società B, in cerca proprio del servizio offerto dalla società A, decide di eseguire una ricerca nel registro dei web service.

- Il servizio viene trovato e con esso la sua descrizione formale.

- Le due società si incontrano e stipulano un accordo di fornitura di servizi attraverso il web firmando un contratto. Questa fase è gestita a livello umano e in effetti non sembra plausibile pensare, anche in un futuro più o meno prossimo, che siano le macchine a occuparsi di questo particolare aspetto,.

- Il cliente integra il web service all'interno del proprio sistema informativo.

- A regime, viene utilizzata una comunicazione XML attraverso Internet per far dialogare in modo automatico i processi delle due aziende.

Questo flusso operativo sposta (in parte) il modo di fare affari, tra le persone alle macchine, rendendo possibile la comunicazione diretta tra i sistemi informatici di due diverse aziende.

Alcuni esempi

Per capire meglio come funzionano e quali sono le potenzialità dei web service, si vedrà ora qualche esempio sia nell'ambito B2B (Business to Business) e B2C (Business to Consumer) che all'interno delle quattro mura dell'azienda.

Business on-line

Un possibile scenario in ambito B2B potrebbe essere quello dove un'azienda ha la necessità di approvvigionarsi di materie prime dai suoi fornitori. Si considerino ad esempio le ipotetiche aziende Tondini Riuniti S.p.A. e Costruzioni S.p.A.

La Tondini Riuniti è una società che vende tondini d'acciaio per l'edilizia e vuole aprire nuovi sbocchi al proprio business sul web. In particolare, vuole mettere a disposizione dei suoi potenziali clienti un modo per potersi approvvigionare senza i costi di una tipica gestione ordini/fatturazione. Il reparto IT della Tondini studia dunque un web service che consente ai clienti di eseguire le ordinazioni del materiale su Internet. Per costruire il componente, si rivolge a un registro di web service e scarica la descrizione di servizio che più si adatta alla sua tipologia applicativa. A questo punto entra in gioco la Costruzioni S.p.A., che desidera instaurare un rapporto di affari per agevolare il più possibile le ordinazioni di tondini per l'edilizia in modo da minimizzare le giacenze di magazzino, riducendo i costi di stoccaggio e massimizzando i profitti.

La Costruzioni ricerca su Internet un fornitore di tondini abilitato ai web service e trova la Tondini Riuniti. Scarica la descrizione tecnica del web service e implementa un programma in grado di comunicare con il servizio. A questo punto, integra il programma di accesso al servizio nel proprio sistema informativo, in modo da eseguire un nuovo ordine ogni volta che la giacenza di magazzino arrivi al livello minimo definito.

Raccolta dati

Un altro scenario di utilizzo potrebbe essere nell'industria, per la raccolta dati. La natura basata su messaggi dei web service rende possibile l'implementazione di questo tipo di sistemi.

Si immagini un'applicazione in campo ambientale. Centrali di raccolta dati sulla qualità dell'aria poste in centri nevralgici della città potrebbe essere dotate di sensori collegati a un microcomputer. Da qui, un'applicazione J2ME (Java2 Micro Edition) potrebbe, ad ogni intervallo di tempo prestabilito, diciamo ogni ora, inviare i dati alla centrale. Le informazioni, codificate in XML, raggiungerebbero poi tramite una connessione Internet o tramite SMS, la centrale di raccolta dati. Qui, un unico web service di raccolta delle informazioni fungerebbe da concentratore, passando le informazioni al programma server che si occuperà di eseguire le elaborazioni statistiche necessarie e di memorizzare le informazioni nel database centrale dell'applicazione.

Integrazione di sistemi

Un'ulteriore applicazione dei web service, questa volta all'interno dell'azienda, è il loro uso per problematiche di integrazione (EAI, Enterprise Application Integration). Tipicamente, l'infrastruttura IT che si è evoluta nel tempo è composta da applicazioni distribuite su piattaforme eterogenee. Programmi COBOL su mainframe spesso costituiscono gli oggetti di business che la presentazione utente e le applicazioni di terze parti devono interfacciare. Spesso ci si scontra con la necessità di portare sul web le applicazioni sviluppate su mainframe e le soluzioni di intergrazione con l'host che sono state applicate in passato sono molteplici. Tipicamente i prodotti commerciali di integrazione hanno sempre proposto un insieme di API proprietarie: con i web service nasce l'opportunità di integrare il mainframe con standard non proprietari. È possibile dunque pensare all'integrazione dei programmi COBOL in XML su HTTP, magari utilizzando un gateway sul lato server. A questo punto, le applicazioni sono in grado di accedere ai programmi in modo agevole, magari passando attraverso un registro di servizi interno per l'individuazione del componente da richiamare.

Il disaccoppiamento che può realizzarsi utilizzando un XML indipendente dalla natura e conformazione dei componenti può portare a una maggiore facilità di reingegnerizzazione del lato mainframe, quando questa possa effettivamente diventare un'opzione. A questo punto lo sviluppatore dell'applicazione non è costretto a imparare l'API di uno specifico prodotto di integrazione ma può rifarsi a un più aperto insieme di specifiche.

Shopping palmare

In un mondo — e in un Paese come il nostro — dove la diffusione dei telefoni cellulari ha raggiunto negli ultimi anni traguardi considerevoli, non è difficile immaginare la diffusione di dispositivi che siano la convergenza degli attuali telefonini e dei computer palmari. Questi dispositivi, anche se dotati di capacità di elaborazione e memoria limitati, possono offrire potenzialità enormi. Ad esempio, un utente potrebbe dedicarsi allo shopping stando comodamente seduto in treno.

Sullo schermo del suo palmare potrebbe accedere a un catalogo di prodotti, riempire il suo carrello della spesa e poi ingaggiare un web service per l'inoltro dell'ordine. La meccanica sarebbe molto simile a quanto succede oggi nei siti web di acquisto on-line, ma la limitata disponibilità di spazio del palmare non consentirebbe l'accesso allo stesso sito web utilizzato con un normale browser. Ecco quindi che un programma apposito potrebbe essere più adeguato, anche se, in entrambi i casi, all'atto dell'ordine — con la necessità di accedere al sistema informativo aziendale — potrebbe a ogni modo essere richiamato un web service.

Agenzia viaggi virtuale

Un ultimo scenario è la citazione di un caso d'uso di un prodotto di HP, sicuramente in anticipo sui tempi, chiamato eSpeak. Questa infrastruttura, concettualmente simile ai web service, si spingeva ancora oltre, prospettando l'integrazione di una serie di servizi e lasciando al software un certo grado di libertà nella scelta dei componenti da integrare.

In questo scenario, trasposto per l'Italia, il signor Rossi desidera fare un viaggio in Sardegna per le vacanze estive. Si connette a Internet e accede al sito web di un'agenzia di viaggi virtuale. Qui inserisce la data di partenza, quella di ritorno e una serie di altre opzioni, come il numero di stelle dell'albergo o la cilindrata della macchina a noleggio che desidera trovare sul luogo.

A questo punto, il software di back-end del sito web comincia ad eseguire una serie di richieste ai web service di diversi fornitori. Interpella il registro dei servizi e identifica le società che offrono traghetti per le isole o che affittano macchine in Sardegna. Contatta Sardinia Ferries e suoi concorrenti per verificare la disponibilità del traghetto e il costo. Confronta i prezzi e sceglie il meno caro ma che risponda alle richieste di qualità del cliente. Verifica la disponibilità dell'albergo, che abbia le caratteristiche richieste e che abbia una stanza libera vista mare per due persone (il signor Rossi vuole fare una vacanza con la moglie). Il back-end del sito va avanti così fino a raccogliere tutte le informazioni necessarie per organizzare il viaggio. Una volta completato il lavoro, presenta al signor Rossi le possibili opzioni. A lui non rimane altro che sceglierne una e dare il numero di carta di credito per il pagamento.

Una volta ottenuti gli estremi, il web service si occupa di formalizzare gli ordini a tutte le parti prenotando i vari beni o servizi richiesti. Ovviamente, nel fare questo, il web service si occuperà, nel caso di eventuali errori, di applicare una rollback distribuita su tutte le parti interessate.

Scenario interessante ma riassumibile così: azzardato allora, impraticabile ancora a oggi. In un futuro lontano, probabile.

Ricerca delle informazioni

Un'altra applicazione dei web service può essere quella relativa alla ricerca delle informazioni strutturate su Internet. Un'applicazione di ricerca potrebbe conoscere le interfacce XML di vari servizi quali la tracciabilità delle spedizioni tramite i corrieri (come UPS o DHL o altri), oppure la visualizzazione di mappe anche per costruire percorsi. Oppure potrebbe collegarsi a un servizio per conoscere in tempo reale la programmazione dei film nei cinema della città, o collegarsi ai servizi delle compagnie aeree per conoscere i voli disponibili per raggiungere un determinato luogo. Fantascienza? Realtà per gli utenti di Apple: la nuova versione del sistema operativo dei Macintosh, Mac OS X 10.3 dispone di uno strumento di ricerca così strutturato (fig. 9.1).

Il programma consente queste particolari ricerche strutturate che hanno la particolarità di non chiamare motori di ricerca tradizionali come Google, ma di accedere a servizi web specifici per la distribuzione di quelle particolari informazioni.

Altri di questi scenari proposti sono concreti e magari sono stati già realizzati. Ad esempio, nel settore bancario, l'integrazione del mainframe tramite XML è una soluzione che è già stata adottata nel recente passato. Anche se alcune volte non sono stati utilizzati i protocolli affermati per i web service ma è stata sviluppata una soluzione proprietaria, gli elementi di base dei web service (Internet e XML) sono stati utilizzati. Altri scenari sono più futuribili: ad oggi i palmari

Figura 9.1 – *L'interfaccia utente di Sherlock*



wireless non sono sicuramente un fenomeno di massa che giustifichi il costo per la creazione e la manutenzione di web service a loro uso e consumo.

Altri scenari ancora sono sicuramente da collocare in un futuro nemmeno troppo vicino: il caso dell'agenzia di viaggi è molto complesso e, oltre alla difficoltà di implementare una logica applicativa di orchestrazione così delicata, sembrano maggiori le difficoltà di carattere organizzativo che nascono quando viene coinvolto un così grande numero di interlocutori.

Ad ogni modo, queste sono le potenzialità che offrono i web service: la loro realizzazione è più o meno vicina.

Tutti gli scenari, comunque, sono tecnicamente realizzabili anche con strumenti tradizionali, sebbene la loro effettiva implementazione possa essere frenata dalle problematiche ad essi legate.

Java e web service

Da più parti si sente accostare ai web Services la piattaforma Java, con svariate buone ragioni. Per prima cosa, la base dei web Services è Internet e XML. Queste due tecnologie si sposano molto bene con la piattaforma Java: gli statunitensi dicono che Java ha un *natural fit* per esse. Per quanto riguarda Internet, Java è nato per interagire con la rete e, sebbene abbia per certi versi effettuato una migrazione, dal client — con le Applet — al server — con la piattaforma J2EE (Java 2 Enterprise Edition) — ha sempre mantenuto un forte legame con Internet e le sue

tecnologie. Un package di base di Java, presente fin dalla versione 1.0 è java.net e dispone di classi per dialogare attraverso le socket e trattare con gli URL. Al di sopra di questo, Java ha poi sviluppato tutta una serie di tecnologie a più alto livello come JavaMail, per il supporto della posta Internet e dei protocolli relativi, le tecnologie server quali Servlet e JSP (Java Server Pages) per l'estensione dei web server, la tecnologia EJB (Enterprise JavaBeans) per portare le applicazioni enterprise su Internet.

Inoltre Java ha una forte convergenza con XML in quanto se il primo è considerato il linguaggio "portabile" (o almeno il più portabile a oggi disponibile), il secondo consente la portabilità delle informazioni. Java ed XML sono dunque il connubio che consente di rendere portabile tutta l'applicazione.

Da qualche parte si sente dire che XML da solo potrebbe essere la soluzione per realizzare applicazioni portabili, ma c'è da considerare che, sebbene l'informazione XML sia in effetti indipendente dalla piattaforma, tutte le volte che questa raggiunge un computer, ha la necessità di essere elaborata. Se il programma che la interpreta non è realizzato in Java, questo dovrà essere reimplementato su tutte le diverse piattaforme.

In secondo luogo è bene chiarire che le tecnologie alla base dei web service non prevedono standard per la realizzazione della logica applicativa dei componenti, e probabilmente non la prevederranno neanche in futuro. Questo perché i web service non sono altro che una "facciata" (*façade*) per componenti applicativi già esistenti.

Chiarito questo aspetto, appare ovvio che la piattaforma a oggi più affermata per l'implementazione dei servizi di business su Internet, J2EE, sia una candidata eccellente all'implementazione dei web service. Implementare i servizi come componenti EJB consente di sfruttare la piattaforma migliore per la realizzazione dei servizi sul server e permette poi la loro presentazione all'esterno come servizi web.

Java Web Services Developer Pack

La Sun ha raccolto le proprie tecnologie a supporto dei web service all'interno del JWSDP (Java Web Services Developer Pack). Questo insieme di tecnologie è un toolkit organico che ruota attorno ad una versione "personalizzata" di Tomcat. Precisamente, le tecnologie presenti all'interno del JWSDP sono:

- Java API for XML Messaging (JAXM);
- Soap with Attachments API for Java (SAAJ);
- Java API for XML Processing (JAXP);
- Java API for XML Registries (JAXR);
- Java API for XML-based RPC (JAX-RPC);
- JavaServer Pages™ Standard Tag Library (JSTL);

- Tomcat;
- Ant;
- strumenti di amministrazione, gestione e deploy delle web application;
- Registry Server UDDI.

Tramite questa distribuzione di Sun è possibile sperimentare e sviluppare praticamente con tutte le tecnologie di base dei web service, come SOAP, WSDL ed UDDI.

Il software è scaricabile dall'indirizzo http://java.sun.com/webservices.

Non solo JWSDP

Altri strumenti degni di nota sono Apache Axis, l'implementazione open source del consorzio Apache nata dalle ceneri dell'implementazione di SOAP di IBM donata diverso tempo fa proprio al consorzio Apache. Il particolare interessante di Axis è il fatto che supporta lo standard in via di definizione JWS (Java Web Service) che consente di creare servizi web senza codifica ma inserendo alcuni metatag di definizione (simili a quelli utilizzati in Javadoc) all'interno del codice che implementa il servizio.

Nella forma più semplice, è possibile creare una semplice classe Java, rinominarla con l'estensione .jws, copiarla sotto Tomcat per fare in modo che il runtime di Axis la riconosca e gli crei intorno un'infrastruttura dinamica che consente di esporre la classe come servizio web.

Un altro strumento interessante è il Borland web Services Developer Kit che non è altro che Axis con l'aggiunta di wizard visuali che semplificano molto l'approccio allo sviluppo dei web service.

Axis è scaricabile all'indirizzo http://xml.apache.org/axis/index.html, mentre il suo predecessore Apache SOAP è disponibile a http://xml.apache.org/soap/index.html.

Comunicare: SOAP e JAXM

Come abbiamo visto prima, la comunicazione è un elemento fondamentale nella visione dei web service e il protocollo affermato che risolve questo aspetto è SOAP.

Acronimo di Simple Object Access Protocol, è uno standard nato in casa Microsoft e supportato in una seconda fase da IBM. Ora è quasi uno standard del W3C, che ne ha raccolto la versione 1.1 proposta dalle due aziende, e che ne sta derivando l'evoluzione (versione 1.2).



Le specifiche di SOAP 1.2 sono definite in due diversi documenti ("Part 1: Messaging Framework" e "Part 2: Adjuncts") a cui è stato aggiunto un terzo documento di introduzione. Sebbene gran parte della logica e dei concetti di SOAP 1.1 sia mantenuta in SOAP 1.2, in quest'ultima versione sono state



applicate diverse piccole variazioni, cambi di nomi e chiarimenti di semantica, che rendono incompatibili le due versioni. Per di più, si noti che la presenza di namespace specifici all'interno dei messaggi è il meccanismo utilizzato da SOAP per garantire che un nodo elabori un messaggio in funzione della specifica versione in cui è stato codificato. Ad esempio, un servizio è in grado, analizzando il messaggio, di verificare se questo è stato codificato secondo le specifiche SOAP 1.1 oppure SOAP 1.2 e attivare le due differenti gestioni, in funzione della versione.
In questo capitolo si considererà solo SOAP 1.1, in quanto è la versione stabile attualmente disponibile e quella supportata dalla piattaforma Java e dalla maggior parte degli strumenti disponibili.

Nelle specifiche SOAP convivono quattro diversi aspetti che affrontano differenti parti del problema:

1. il modello di scambio dei messaggi;
2. l'encoding SOAP;
3. il collegamento tra SOAP ed HTTP;
4. l'utilizzo di SOAP come meccanismo di RPC.

Il modello di scambio dei messaggi definisce un modo standard per definire come strutturare messaggi XML che saranno scambiati tra i diversi nodi, come vengono gestiti gli errori e come si devono chiamare i tag XML che contengono le informazioni.

L'encoding SOAP fornisce una specifica (opzionale), per definire come strutture dati anche complesse, quali enumerazioni e vettori sparsi, vengono rappresentate in XML. È basata su una prima bozza delle specifiche XML Schema.

Il collegamento con HTTP, invece, definisce il modo in cui i messaggi SOAP debbano essere veicolati tramite questo protocollo di filo e come devono essere utilizzate le intestazioni HTTP e i codici di ritorno per l'invio di messaggi SOAP.

Infine, il modello RPC definisce il modo in cui SOAP può essere utilizzato per rappresentare chiamate a procedure remote. Le specifiche SOAP consentono infatti di realizzare applicazioni di semplice messaggistica o più evolute applicazioni distribuite, anche se mancano i dettagli per aspetti legati alla sicurezza e alla garbage collection distribuita.

Anatomia del messaggio

La comunicazione tramite SOAP avviene tramite lo scambio di messaggi strutturati in modo specifico comprendente elementi obbligatori ed elementi opzionali. Il tag principale che racchiude tutti gli altri è `Envelope` (busta). Dentro questo sono presenti `Header` (intestazione) e `Body` (cor-

po). Il primo è opzionale, mentre nel corpo sono presenti tutte le informazioni applicative che il nodo vuole inviare.

Esempio 9.1 – *Una richiesta SOAP.*

```
<?xml version="1.0"?>

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/>

<SOAP-ENV:Body>
  <ns1:getRate xmlns:ns1="urn:xmethods-CurrencyExchange">

    <country1>England</country1>
    <country2>Japan</country2>

  </ns1:getRate>
</SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

Nell'esempio 9.1 è presente un semplice messaggio SOAP. I tag Envelope e Body appartengono al namespace http://schemas.xmlsoap.org/soap/envelope. L'utilizzo di SOAP-ENV è semplicemente una convenzione: al suo posto si sarebbe potuto utilizzare un qualsiasi nome simbolico, come ad esempio ns1, utilizzato nel corpo del messaggio.

Come si può notare, all'interno del tag Body è definito il messaggio applicativo: nel tag getRate sono presenti i tag country1 e country2. Questo blocco SOAP potrebbe essere un messaggio utilizzato in una applicazione che si occupi di fornire il tasso di conversione tra diverse valute.

L'utilizzo dei namespace è importante: consente di essere sicuri che non ci siano conflitti di nomi: come due classi Java con lo stesso nome possono convivere nello stesso programma se appartengono a package differenti, anche in SOAP possono coesistere tag con lo stesso nome, purché questi appartengano a namespace diversi.

La risposta SOAP (esempio 9.2) è strutturata in modo similare, con tag Envelope e Body e con una risposta applicativa all'interno di quest'ultimo.

Esempio 9.2 – *Una risposta SOAP.*

```
<?xml version="1.0"?>

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/>

<SOAP-ENV:Body>
```

```
    <ns1:getRateResponse xmlns:ns1="urn:xmethods-CurrencyExchange">

      <return>154.9423</return>

    </ns1:getRateResponse>
  </SOAP-ENV:Body>

  </SOAP-ENV:Envelope>
```

Per convenzione, il nome del tag utilizzato nella risposta è lo stesso della richiesta, a cui è aggiunta la parola Response. Nell'esempio, il tag applicativo contiene il valore di ritorno della chiamata al servizio, e cioè il tasso di conversione tra le due divise specificate.

I due blocchi SOAP visti in precedenza presuppongono un modello di interazione tra due nodi di tipo request/response: a ciascuna richiesta, il ricevente fornisce una risposta al chiamante. SOAP è utilizzabile anche in modalità *oneway* (a senso unico, detta anche *fire&forget*), dove il client si limita a inviare un messaggio al server, senza che questo formuli una risposta. Queste due modalità verranno viste più in dettaglio nella parte dedicata a WSDL, che tra l'altro, ne definisce di ulteriori.

SOAP su HTTP

Le regole per utilizzare SOAP su HTTP ben si sposano con quest'ultimo protocollo, in quanto non vengono aggiunti elementi estranei, ma vengono sfruttate le semantiche già esistenti. In particolare, le regole di mappatura possono essere così riassunte:

- il Content-Type deve essere text/xml;

- l'intestazione HTTP deve riportare il campo SOAPAction. Questo indica al server web o al firewall l'intento del messaggio, per eventuali operazioni di filtro.

Entrambe queste indicazioni sono obbligatorie.

L'invio di una richiesta dal client al server, avviene tramite una operazione di POST, come si evince dall'esempio 9.3. Qui è possibile vedere l'intestazione HTTP di una classica richiesta SOAP, con l'indicazione corretta del Content-Type e di SOAPAction.

Esempio 9.3 – *Richiesta SOAP veicolata sul protocollo http.*

```
POST /StockQuote HTTP/1.1
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "http://electrocommerce.org/abc#MyMessage"

<SOAP-ENV:Envelope...
```

Per quanto concerne la risposta (esempio 9.4), il client deve fare affidamento sul codice HTTP di risposta per capire se tutto è andato bene o se si è verificato qualche problema.

Esempio 9.4 – *Risposta SOAP veicolata sul protocollo HTTP.*

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope...
```

Nel caso dell'esempio 9.4, il server ha restituito il codice 200, che indica la corretta elaborazione della risposta. In entrambi i casi, ovviamente, il campo Content-Length conterrà la corretta dimensione del corpo del messaggio SOAP.

Se qualcosa va male

Se avviene qualche errore in fase di elaborazione del messaggio SOAP, il server può inviare come risposta, al posto della normale risposta SOAP, uno specifico messaggio di errore, detto Fault. Questo definisce, in modo standardizzato, le specifiche dell'errore verificatosi. I contenuti del tag Fault possono essere faultcode, faultstring, faultactor, detail, che identificano, nell'ordine, il codice dell'errore, la descrizione dell'errore, l'entità che lo ha generato e opzionalmente i dettagli dell'errore, strutturati come un blocco XML applicativo. Nell'esempio 9.5 è presente un di Fault SOAP.

Esempio 9.5 – *Esempio di Fault SOAP.*

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Body>
      <SOAP-ENV:Fault>
        <faultcode>SOAP-ENV:Server</faultcode>
        <faultstring>Errore server</faultstring>
        <detail>
          <e:myfaultdetails xmlns:e="Some-URI">
          <message>
            Si è verificato un errore applicativo
          </message>
          <errorcode>
            1001
          </errorcode>
        </e:myfaultdetails>
      </detail>
```

```
      </SOAP-ENV:Fault>
    </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>
```

Il codice dell'errore è una stringa, composta da parole concatenate, come ad esempio Client.Authentication. Lo scopo delle specifiche è quello di fornire un meccanismo che sia più flessibile ed espandibile rispetto ai più classici codici di errore numerici, come quelli utilizzati in HTTP.

I prefissi che possono essere utilizzati nei codici di errore sono predefiniti, e sono:

- VersionMismatch: indica che il namespace sul tag Envelope che identifica la versione ha un valore diverso da quello atteso;
- MustUnderstand : l'elaborazione obbligatoria di una parte dell'intestazione ha avuto esito negativo;
- Client: può indicare errori in merito alla strutturazione del messaggio di richiesta;
- Server: indica l'impossibilità di elaborare una risposta, anche a fronte di richieste correttamente strutturate, ad esempio nel caso di inconsistenze interne del componente server.



Un utilizzo interessante del tag detail è quello che prevede la sua valorizzazione con lo stack trace della chiamata che ha generato l'errore, in modo da semplificare il debug dell'applicazione. Questa informazione, infatti, non è obbligatoriamente da mostrare all'utente, ma l'applicazione la potrebbe mostrare solo a sviluppatori e sistemisti fornendo un'informazione preziosa per individuare l'errore.

Il collegamento con il protocollo HTTP definisce che il blocco SOAP Fault debba essere veicolato con un messaggio con codice di errore 500 (utilizzato in HTTP per identificare gli errori del server), come si evince dall'esempio 9.6.

Esempio 9.6 – *Fault SOAP veicolato tramite HTTP.*

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
```

Encoding

Il concetto di encoding riguarda le informazioni applicative contenute nel tag Body: definisce il modo in cui i dati vengono rappresentati in XML. Le specifiche SOAP lasciano una certa libertà, ma suggeriscono l'uso dell'encoding SOAP. Altri, come Apache, hanno storicamente proposto un econding più semplice, denominato encoding letterale.

Ecoding letterale

Questo tipo di encoding è semplicemente un blocco XML applicativo appartenente opzionalmente a un determinato namespace (esempio 9.7).

Esempio 9.7 – *Un encoding letterale.*

```
<Contatti>
  <Persone>
    <Persona id="1">
      <Nome>Mario</Nome>
      <Cognome>Rossi</Cognome>
    </Persona>
    <Persona id="2">
      <Nome>Giuseppe</Nome>
      <Cognome>Verdi</Cognome>
    </Persona>
  </Persone>
</Contatti>
```

Anche i primi esempi SOAP visti in precedenza (esempi 9.1 e 9.2), utilizzano un encoding letterale, però con uno specifico namespace. Come si nota, la codifica è molto semplice, ma non sono presenti informazioni sui tipi di dati.

Encoding SOAP

L'encoding SOAP definisce regole (derivate da una prima versione di XML Schema), per rappresentare il tipo di dato utilizzato nel blocco dati applicativo, mutuando le tipologie dai più diffusi linguaggi di programmazione e dai tipi di dati utilizzati nei database.

Le strutture complesse, come i dati compositi (ad esempio le *struct* del linguaggio C), trovano quindi uno standard per la loro serializzazione in XML. Sono previsti diversi tipi di dati, come:

- tipi semplici (numeri interi, reali, stringhe, etc);
- enumerazioni;
- array di bytes (base64);

- strutture;
- array.

La gestione degli array in particolare è molto potente, in quanto consente anche di specificare array sparsi, dove cioè, gli indici possono anche non essere contigui (ad esempio possono essere presenti solo gli elementi 1, 3, 4, 11).

L'encoding SOAP è più stringente rispetto all'encoding letterale. Ciò consente di rendere più facile l'interoperabilità delle informazioni trasferendo, insieme ai dati, anche metainformazioni che ne descrivono il tipo. Ad esempio, è possibile capire direttamente dal flusso XML se un determinato tag è di tipo stringa o numerico.

Per esempio, nel blocco XML:

```
<X1>12</X1>
```

il valore 12 potrebbe essere numerico, ma il tag potrebbe anche potenzialmente essere di tipo alfanumerico). Con l'encoding SOAP avremmo

```
<X1 xsd:type="String">12</X1>
```

e il contenuto dell'attributo type chiarisce che il tipo di dato è alfanumerico. Sempre in merito agli esempi 9.1 e 9.2, è possibile codificarli con l'encoding SOAP, come mostrato negli esempi 9.8 e 9.9.

Esempio 9.8 – *Una chiamata codificata con l'encoding SOAP.*

```
<?xml version="1.0"?>

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">

<SOAP-ENV:Body>
  <ns1:getRate xmlns:ns1="urn:xmethods-CurrencyExchange"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">

    <country1 xsi:type="xsd:string">England</country1>
    <country2 xsi:type="xsd:string">Japan</country2>

  </ns1:getRate>
</SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

Esempio 9.9 – *Una risposta codificata con l'encoding SOAP.*

```
<?xml version="1.0"?>

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">

<SOAP-ENV:Body>
  <ns1:getRateResponse xmlns:ns1="urn:xmethods-CurrencyExchange"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">

    <return xsi:type="xsd:float">154.9423</return>

      </ns1:getRateResponse>
</SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

Si può notare in questi esempi (9.8 e 9.9), come ciascun tag applicativo specifichi anche un attributo che identifica il tipo di dato.

SOAP RPC

Le specifiche SOAP definiscono anche gli standard per utilizzare messaggi SOAP per realizzare semplici applicazioni distribuite di tipo Remote Procedure Call (RPC). In particolare, le chiamate sono descritte all'interno del tag body e i parametri e il valore di ritorno sono modellati come strutture, anche se la scelta dell'encoding è libera. In caso di errore, questo è codificato come Fault SOAP.



La differenza tra semplici messaggi SOAP e SOAP-RPC è che nel primo caso si parla di generici messaggi di notifica che possono veicolare meri contenuti informativi (come ad esempio, un intero documento XML da inserire in un database). In caso di SOAP-RPC, nel corpo del messaggio SOAP è presente una formalizzazione in XML di una chiamata a metodo, quindi tipicamente le informazioni sul nome del metodo e sui valori dei parametri.

SOAP-RPC verrà visto più nel dettaglio nella prossima sezione, tramite un esempio pratico.

JAXM e SAAJ

L'implementazione standard di SOAP in Java è definita dalle due specifiche JAXM (Java API for XML Messaging) e SAAJ (SOAP w/Attachments API for Java). Le prime definiscono le

chiamate standard per una generica messaggistica XML (package javax.xml.messaging); le seconde sono l'implementazione del modello informativo di SOAP 1.1 in Java (package javax.xml.soap).



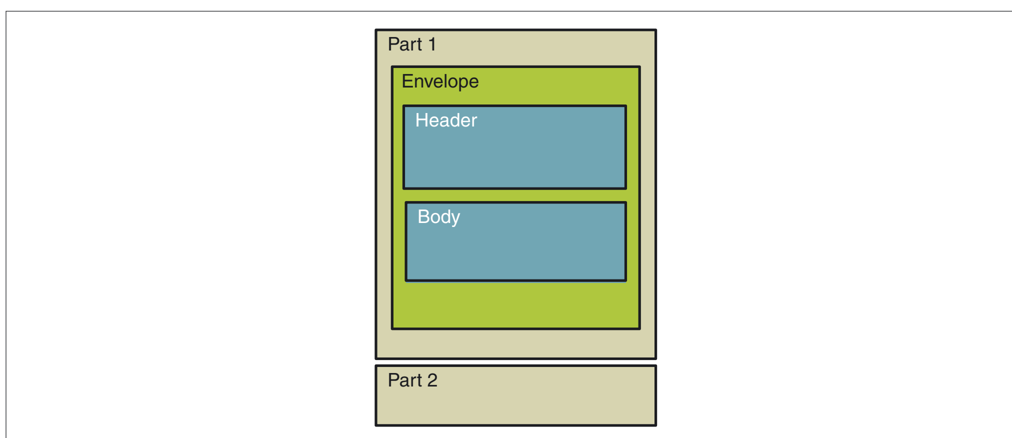
Perché SOAP w/Attachments ("con allegati")? In realtà la piattaforma Java supporta, oltre a SOAP 1.1, anche un altro insieme di specifiche che indica come utilizzare MIME per inviare messaggi binari (come immagini o file .jar) come parte di un allegato a un messaggio SOAP. Questo aspetto entrerà maggiormente in gioco quando si parlerà di JAX-RPC e WSDL.

Le JAXM supportano le due tipologie di interazione di SOAP (richiesta/risposta e a senso unico), fornendo una infrastruttura per formulare e ricevere messaggi SOAP. Quest'ultima possibilità è però prerogativa di applicazioni che girino all'interno di web container J2EE in quanto il supporto è costruito attorno alla tecnologia delle servlet. Sempre queste ultime consentono di supportare l'invio di messaggi *asincroni*, chiamate che non bloccano il client in attesa di una risposta.



Purtroppo, anche le SAAJ hanno capacità di messaging (tramite le classi SOAPConnection e SOAPConnectionFactory), e la separazione tra le due API non è dunque netta: le due sono infatti nate insieme e sono state separate in un secondo tempo per consentire ad altre tecnologie Java come JAX-RPC la condivisione di codice per SOAP. Il resto del testo potrebbe dunque riferirsi in modo generico a JAXM intendendo sia queste che SAAJ.

Figura 9.2 – *La struttura di un messaggio SOAP con allegati.*



Object model SOAP

Le classi e le interfacce di SAAJ consentono una completa modellazione di un messaggio SOAP complesso. In particolare esistono entità per rappresentare i singoli elementi, come Body, Header, Envelope e la sezione MIME. Ciascuna entità possiede metodi specifici per manipolare il tag di appartenenza, ad esempio per specificare la tipologia di encoding o per impostare i namespace. Lo schema di un messaggio SOAP con allegati è descritto in fig. 9.2.

Le singole entità sono tra loro collegate tramite opportuni metodi. Ad esempio, a partire da un messaggio SOAP di tipo SOAPMessage è possibile ottenere riferimenti ai singoli elementi (come Body, Header, Envelope), come si evince dall'esempio 9.10.

Esempio 9.10 – *Modello ad oggetti di SAAJ.*

```
SOAPPart sp = msg.getSOAPPart();
SOAPEnvelope envelope = sp.getEnvelope();
SOAPHeader hdr = envelope.getHeader();
SOAPBody bdy = envelope.getBody();
```



Per ottenere nuovi messaggi SOAP con JAXM è necessario però partire da una factory, sia essa ottenuta tramite la classe MessageFactory, nel caso di programmi standalone, oppure attraverso un "provider" se l'applicazione è in funzione in un ambiente J2EE ed il tipo di messaggio richiesto è di tipo asincrono. Si sarebbe potuto creare un messaggio SOAP semplicemente concatenando delle stringhe, ad esempio utilizzando StringBuffer e il risultato sarebbe stato il medesimo. Lo scopo finale di SAAJ infatti, è quello di produrre un flusso XML ben formattato. Ovviamente, l'utilizzo di API specifiche permette di organizzare meglio il codice e renderlo più manutenibile, oltre a semplificare operazioni di creazione particolarmente complesse. Inoltre, API specifiche semplificano molto le operazioni di parsing della risposta, in quanto forniscono chiamate già pronte per accedere ai singoli elementi di un messaggio SOAP.

Nel primo caso, è necessario chiamare il metodo newInstance() della classe MessageFactory.

```
MessageFactory mf = MessageFactory.newInstance();
```

La factory possiede il metodo createMessage() che consente la creazione di nuovi messaggi posizionati nella part MIME principale:

```
SOAPMessage msg = mf.createMessage();
```

Inizialmente i tag sono vuoti: il loro contenuto andrà valorizzato successivamente operando direttamente tramite gli appositi oggetti.



La gerarchia di classi di SAAJ è strutturata a partire da Node, superinterfaccia di SOAPElement, da cui derivano le altre, ad esempio SOAPHeader e SOAPBody. Non si confonda questo Node con quello di JAXP in quanto, sebbene esprimano concetti simili (nodi di un albero) appartengono a gerarchie di classi diverse.

Per costruire un completo messaggio SOAP come quello visto nell'esempio 9.8, è necessario chiamare gli opportuni metodi per esplicitare ogni singolo dettaglio XML da includere nell'output. Si consideri l'esempio 9.12: contiene un estratto del programma ExchangeClient.java, un client per il servizio di cambio valuta che tratta con i messaggi SOAP visti negli esempi 9.8 e 9.9.



Il servizio di cambio valuta è disponibile su XMethods (http://www.xmethods.com). Questo sito raccoglie una vasta collezione di web service costruiti sulle più disparate piattaforme: da Java a .NET, da Apache SOAP ad Axis, da Perl a Kylix. Oltre al catalogo dei servizi presenti su Internet, fornisce anche alcuni servizi di prova, come ad esempio quello preso qui in esame. Una funzione interessante del sito è l'analizzatore di WSDL (la "descrizione del servizio", si veda la sezione seguente di questo capitolo per ulteriori dettagli) che consente di visualizzare le informazioni relative alle funzioni, ai parametri e ai valori di ritorno di un servizio, in forma molto leggibile.

Come si può notare dal listato, viene fatto uso della factory per ottenere un nuovo messaggio vuoto e da questo vengono estratti i riferimenti a ciascun elemento del messaggio SOAP. Questi sono poi oggetto di manipolazione, in particolare:

- vengono aggiunte le dichiarazioni dei namespace alla busta (Envelope), tramite il metodo addNamespaceDeclaration();
- viene impostato il tipo di encoding sul tag getRate a SOAP encoding;
- vengono esplicitamente impostati gli attributi type sugli elementi country.

Si noti che, per creare i tag, viene fatto uso del metodo createName() presente in SOAPEnvelope. Questo metodo è disponibile in due forme: con un solo parametro, per creare un tag semplice; con tre parametri per creare un tag che appartenga ad un namespace, come si può notare nella riga di codice che crea l'elemento SOAPBodyElement.

Esempio 9.11 – *Namespace, encoding e attributi con JAXM*

```
//Il messaggio qui è già stato creato
```

```
SOAPPart    sp  = msg.getSOAPPart();
SOAPEnvelope  env = sp.getEnvelope();
SOAPHeader    hdr = env.getHeader();
SOAPBody    bdy = env.getBody();

String xsi = "http://www.w3.org/2001/XMLSchema-instance";
env.addNamespaceDeclaration("xsi", xsi);
env.addNamespaceDeclaration("xsd", "http://www.w3.org/2001/XMLSchema");
env.setEncodingStyle("http://schemas.xmlsoap.org/soap/encoding/");

javax.xml.soap.Name xsiTypeString = env.createName("type", "xsi", xsi);

SOAPBodyElement gltp = bdy.addBodyElement(env.createName("getRate",
"ns1",
"urn:xmethods-CurrencyExchange")
);

SOAPElement e1 =
gltp.addChildElement(env.createName("country1")).addTextNode(divisaIniziale);
e1.addAttribute( xsiTypeString, "xsd:string" );
```

Se eseguito, questo codice produce in output lo stesso flusso XML visto nell'esempio 9.8.

Le API per il messaging

Una volta costruito il messaggio SOAP è necessario inviarlo. Per fare questo è possibile utilizzare due approcci diversi, con o senza un "provider di connessioni". Il provider è una sorta di server di messaggistica che vive all'interno di contenitori J2EE e che si occupa dell'instradamento del messaggio al posto del runtime di JAXM. Questo meccanismo consente l'implementazione di sistemi asincroni e l'utilizzo di profili che possono interagire con il messaggio prima che questo venga effettivamente inviato al reale destinatario.

In alternativa, è possibile utilizzare le API SAAJ che forniscono chiamate sincrone, tramite le classi SOAPConnectionFactory e SOAPConnection. In tab. 9.1 sono riassunte le differenze tra connessioni SAAJ e tramite provider.

Per inviare un messaggio senza provider è sufficiente chiamare il metodo call() sulla connessione SOAPConnection, passandogli l'indirizzo del servizio (chiamato endpoint). JAXM definisce la classe URLEndpoint per rappresentare un endpoint rappresentato come URL. È possibile vedere quanto detto nell'esempio 9.12.

Esempio 9.12 – *Invio di un messaggio SOAP tramite connessione standalone.*

```
URLEndpoint urlEndpoint = new URLEndpoint("http://services.mokabyte.it:80/endpoint");

SOAPConnectionFactory scf = SOAPConnectionFactory.newInstance();
```

Tabella 9.1 – *Riassunto delle caratteristiche JAXM per connessione con e senza provider.*

Caratteristica	***Con*** **provider**	***Senza*** **provider**
Factory	ProviderConnectionFactory (solitamente ottenuta da un rifermento JNDI)	SOAPConnectionFactory
Connessione	ProviderConnection	SOAPConnection
Tipo chiamata	Asincrona	Sincrona
Bidirezionalità	One-way (a senso unico)	Request/response (richiesta/risposta), ma nel caso il server non fornisca risposta, l'oggetto SOAPMessage ritornato è vuoto e liberamente scartabile.
Indirizzamento	Configurato nel server e su base di factory	Da specificare in fase di chiamata

```
SOAPConnection con = scf.createConnection();
SOAPMessage reply = con.call( msg, urlEndpoint );
```

Con un provider, l'esempio è simile ma non coinvolge l'uso degli endpoint. Innanzitutto si parte da un contesto JNDI tramite InitialContext(), poi si ottiene da questo la factory e da questa la connessione. L'invio avviene tramite la chiamata al metodo send().

Esempio 9.13 – *Utilizzo del provider di connessioni.*

```
Context ctx = new InitialContext();
ProviderConnectionFactory pcf = (ProviderConnectionFactory)ctx.lookup("ProviderXYZ");
ProviderConnection con = pcf.createConnection();
con.send( msg );
```

I due provider presenti in JAXM 1.1 riguardano ebXML-M e SOAP-RP. Il protocollo di trasporto ebXML-M (utilizzato nella tecnologia ebXML, si veda la fine del capitolo per ulteriori informazioni), utilizza SOAP come base, ma richiede la popolazione di particolari informazioni nel tag header del messaggio. Con il profilo relativo, JAXM consente di gestire le maggiori informazioni nell'intestazione. SOAP-RP (ora WS-Routing) è invece uno standard per il routing dei messaggi SOAP che inserisce, nell'intestazione del messaggio, un blocco XML che definisce tutto il percorso che il messaggio dovrà seguire. In questo scenario, infatti, il messaggio non viene semplicemente trasferito dal client al server, ma da questo viene instradato per raggiungere più server diversi fino ad arrivare al destinatario finale. Ciò può essere utile per eseguire operazioni intermedie distribuite sul messaggio: come un controllo antivirus, un controllo di parità o di consistenza. Tutte queste operazioni potrebbero essere svolte in nodi diversi e specializzati.

Per supportare diversi profili, è necessario implementare factory di messaggi personalizzate. Le implementazioni di riferimento di Sun per ebXML-M e SOAP-RP sono comprese nei package com.sun.xml.messagin.jaxm.ebxml e com.sun.xml.messagin.jaxm.soaprp.

Un client per il servizio cambio valuta

A questo punto è semplice mettere insieme i pezzi e creare un semplice client per la fruizione del servizio di cambio valuta. L'implementazione di servizio a cui si accederà è quello di prova di XMethods, disponibile all'indirizzo http://services.xmethods.net:80/soap. Nell'esempio, l'endpoint specifico viene costruito estraendo l'URL da una proprietà di sistema chiamata endpoint. Questa viene passata dallo script di Ant allegato al codice e può assumere il valore visto prima, nel caso si desideri accedere al vero e proprio servizio XMethods, oppure un indirizzo locale, in caso si voglia testare l'implementazione della parte server del servizio illustrata nel paragrafo successivo.

Il codice completo del client è presente nell'esempio 9.14: si riscontrerà quanto già visto nei paragrafi precedenti in merito al modello a oggetti SOAP e alle chiamate per la messaggistica.

Esempio 9.14 – *ExchangeClient.java: un client di esempio del servizio Currency-Exchange.*

```
import java.net.*;
import java.io.*;
import java.util.*;

import javax.xml.messaging.*;
import javax.xml.soap.*;

class ExchangeClient {

  public ExchangeClient() {
    sendMessage( "us", "euro" );
  }

  protected void sendMessage( String divisaIniziale, String divisaFinale ) {
    try {
      MessageFactory mf   = MessageFactory.newInstance();
      SOAPMessagemsg = mf.createMessage();
      SOAPPart    sp   = msg.getSOAPPart();
      SOAPEnvelope  env = sp.getEnvelope();
      SOAPHeader    hdr = env.getHeader();
      SOAPBody     bdy = env.getBody();

      String xsi = "http://www.w3.org/2001/XMLSchema-instance";
      env.addNamespaceDeclaration("xsi", xsi);
      env.addNamespaceDeclaration("xsd", "http://www.w3.org/2001/XMLSchema");
      env.setEncodingStyle("http://schemas.xmlsoap.org/soap/encoding/");

      javax.xml.soap.Name xsiTypeString = env.createName("type", "xsi", xsi);

      SOAPBodyElement gltp = bdy.addBodyElement(env.createName("getRate", "ns1",
```

```
                                                    "urn:xmethods-CurrencyExchange"));

    SOAPElement e1 =
    gltp.addChildElement(env.createName("country1")).addTextNode(divisaIniziale);
    e1.addAttribute( xsiTypeString, "xsd:string" );

    SOAPElement e2 =
    gltp.addChildElement(env.createName("country2")).addTextNode(divisaFinale);
    e2.addAttribute( xsiTypeString, "xsd:string" );

    FileOutputStream sentFile = new FileOutputStream("sent.xml");
    msg.writeTo(sentFile);
    sentFile.close();

    URLEndpoint endpoint = new
      URLEndpoint(System.getProperty("endpoint"));

    SOAPConnectionFactory scf = SOAPConnectionFactory.newInstance();
    SOAPConnection conn = scf.createConnection();
    SOAPMessage response = conn.call( msg, endpoint );

    if( response != null ) {
      //L'intestazione XML viene gi‡ messa da sola
      FileOutputStream replyFile = new FileOutputStream("reply.xml");
      response.writeTo(replyFile);
      replyFile.close();

      sp = response.getSOAPPart();
      env = sp.getEnvelope();
      bdy = env.getBody();
      Iterator ii = bdy.getChildElements();
      while( ii.hasNext() ) {
        SOAPElement e = (SOAPElement)ii.next();
        printSOAPElement( e );

        Iterator kk = e.getChildElements();
        while( kk.hasNext() ) {
          SOAPElement ee = (SOAPElement)kk.next();
          printSOAPElement( ee );
        }
      }
    }

  } catch( Exception e ) {
```

```
      e.printStackTrace();
    }
  }

  protected void printSOAPElement( SOAPElement e ) {
    System.out.print( e.getElementName().getLocalName() );
    System.out.print( " = " );
    System.out.println( e.getValue() );
  }

  public static void main( String[] args ) {
    new ExchangeClient();
  }
}
```

Un elemento degno di attenzione è legato alle modalità di estrazione delle informazioni dal messaggio: ogni sottoclasse di SOAPElement dispone del metodo getChildElements(), il quale fornisce un iteratore che elenca i nodi figli dell'elemento attuale:

```
Iterator ii = bdy.getChildElements();
while( ii.hasNext() ) {
SOAPElement e = (SOAPElement)ii.next();
}
```

Unitamente al codice descritto in questo capitolo, è presente un file di compilazione per lo strumento Ant, build.xml. Per compilare ed eseguire il codice qui descritto, è necessario indicare ad Ant di eseguire il target client, che indica di compilare ed eseguire la sola parte client (il file build.xml è utilizzato nel resto del paragrafo su JAXM per gli altri esempi). Per lanciare il client, utilizzare il seguente comando:

```
Ant client
```



Attenzione a due aspetti: all'inizio del file build.xml sono presenti alcune proprietà che indicano il percorso di installazione del JWSDP e del codice di esempio. È necessario modificare questi parametri affinché riflettano i reali percorsi in uso; inoltre è necessario fare attenzione alla proprietà endpoint specificata nel file: indica al programma l'indirizzo del web service a cui collegarsi.

Si consideri infatti il seguente frammento di codice dal file ExchangeClient.java:

```
URLEndpoint endpoint = new URLEndpoint(System.getProperty("endpoint"));
```

Tabella 9.2 – *Le tipologie di servlet di JAXM*

ReqRespListener	OnewayListener
SOAPMessage onMessage(SOAPMessage msg)	void onMessage(SOAPMessage msg)

L'indirizzo dell'endpoint è acquisito dalla proprietà di sistema chiamata endpoint specificata nel file build.xml. In caso di accesso al servizio di prova di XMethods, è necessario dunque attivare la seguente riga:

```
<property name="endpoint" value="http://services.xmethods.net:80/soap" />
```

Un server per implementare il servizio di cambio valuta

Oltre alla realizzazione dei client, JAXM e SAAJ consentono la costruzione di web service basati su SOAP. Attualmente è supportato solo il protocollo SOAP su HTTP, infatti l'infrastruttura si basa sulla servlet JAXMServlet che a sua volta deriva da HttpServlet. Le servlet che diverranno servizi web devono poi implementare una specifica interfaccia, in funzione del fatto che queste ritornino o meno una risposta al client. In questo caso l'interfaccia da implementare è ReqRespListener; in caso di servlet solo riceventi, l'interfaccia è OnewayListener. In particolare, le due interfacce definiscono il metodo onMessage, il cui corpo deve essere implementato nella servlet che implementa il servizio. In tab. 9.2 sono presenti le due firme di onMessage() per le due interfacce.

Per capire come realizzare un servizio con JAXMServlet, verrà ora preso in esame un esempio che implementa proprio il servizio di cambio valuta utilizzato nel paragrafo precedente; ovviamente non sarà presente logica applicativa, ma verrà ritornata una semplice costante numerica.

La struttura della servlet è inversa rispetto al client visto sopra: prima è necessario decodificare il messaggio ed estrarne i contenuti formativi, in seguito è possibile formulare un messaggio di risposta valorizzando gli opportuni tag con le informazioni che il client si aspetta.

Il codice completo è presente nell'esempio 9.15.

Esempio 9.15 – *ExchangeServer.java: un esempio di servlet SOAP.*

```
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import javax.xml.messaging.*;
import javax.xml.soap.*;

public class ExchangeServlet extends JAXMServlet implements ReqRespListener {

  public void init(ServletConfig servletConfig) throws ServletException {
```

```
    super.init(servletConfig);
  }

  public SOAPMessage onMessage(SOAPMessage message) {
    SOAPMessage response = null;
    String c1 = null, c2 = null;
    System.out.println("Messaggio ricevuto");

    try {
      message.writeTo(System.out);

      SOAPPart        sp  = message.getSOAPPart();
      SOAPEnvelope  env = sp.getEnvelope();
      SOAPHeader    hdr = env.getHeader();
      SOAPBody       bdy = env.getBody();

      Iterator ii = bdy.getChildElements();
      while( ii.hasNext() ) {
        SOAPElement e = (SOAPElement)ii.next();

        Iterator kk = e.getChildElements();
        while( kk.hasNext() ) {
          SOAPElement ee = (SOAPElement)kk.next();
          String name = ee.getElementName().getLocalName();

          if( name.equals("country1") ) {
            c1 = ee.getValue();
          }
          if( name.equals("country2") ) {
            c2 = ee.getValue();
          }
        }
      }

      float f = getRate( c1, c2 );
      response = createResponse( f );

      } catch(Exception e) {
      e.printStackTrace();
      }
    return response;
    }
```

```
    protected float getRate( String c1, String c2 ) {
      return (float)10.55;
    }

    protected SOAPMessage createResponse( float f ) throws SOAPException {
      MessageFactory mf  = MessageFactory.newInstance();
      SOAPMessage   msg = mf.createMessage();
      SOAPPart   sp  = msg.getSOAPPart();
      SOAPEnvelope  env = sp.getEnvelope();
      SOAPHeader    hdr = env.getHeader();
      SOAPBody    bdy = env.getBody();

      String xsi = "http://www.w3.org/2001/XMLSchema-instance";
      env.addNamespaceDeclaration("xsi", xsi);
      env.addNamespaceDeclaration("xsd", "http://www.w3.org/2001/XMLSchema");
      env.addNamespaceDeclaration("soapenc", "http://schemas.xmlsoap.org/soap/encoding/");
      env.setEncodingStyle("http://schemas.xmlsoap.org/soap/encoding/");

      javax.xml.soap.Name xsiTypeString = env.createName("type", "xsi", xsi);

      SOAPBodyElement gltp = bdy.addBodyElement(env.createName("getRateResponse",
                                                   "ns1", "urn:xmethods-CurrencyExchange"));

      SOAPElement e1 = gltp.addChildElement(env.createName("Result")).addTextNode(""+f);
      e1.addAttribute( xsiTypeString, "xsd:float" );

      return msg;
    }
  }
```

Come si può notare osservando il codice, il messaggio SOAP ricevuto dalla servlet, già trasformato in un oggetto SOAPMessage, viene scomposto nelle sue parti fondamentali dagli opportuni metodi getter, e analizzato alla ricerca dei tag country, che costituiscono i parametri di chiamata. Questa parte funge da adattatore tra il messaggio SOAP e l'implementazione Java del servizio.

Questo è infatti implementato dal metodo getRate() della servlet:

```
protected float getRate( String c1, String c2 ) {
  return (float)10.55;
}
```

Ovviamente, implementazioni più complesse della mera restituzione di una costante potrebbero risiedere in componenti esterni alla servlet, allo scopo di meglio separare le competenze di ciascun modulo.

Il risultato prodotto dal metodo sopra citato viene costruito dal metodo createResponse() che, come si può notare osservando il codice, contiene le classiche chiamate JAXM per la costruzione di un messaggio SOAP. L'elemento XML che contiene il risultato effettivo è Result, che è di tipo xsd:float.

Per compilare ed eseguire il codice server è possibile utilizzare il target server del file build.xml di Ant. Il comando:

```
Ant server
```

si occupa infatti di compilare la servlet, creare il file WAR necessario alla sua installazione in un web container, e alla sua installazione in Tomcat. In merito a quest'ultimo passaggio è necessario fare attenzione a due aspetti: per prima cosa è indispensabile utilizzare la versione di Ant fornita con il JWSDP, in quanto dispone delle funzionalità aggiuntive necessarie a operare con il web container di Apache; in secondo luogo, è necessario modificare i parametri di autenticazione a Tomcat, username e password, affinché riflettano quelli effettivamente specificati in fase di installazione del JWSDP.

Il file build.xml contiene anche un task per la rimozione del servizio da Tomcat, in caso sia necessario modificarlo e reinstallarlo: questo task si chiama remove.

In caso di errore?

Ma cosa succede se il servizio va in errore, magari perché gli viene passato un dato di input erroneo, come una nazione inesistente? Il servizio di XMethods non ha una gestione degli errori particolarmente evoluta: in caso venga passato un parametro non valido, il componente solleva un'eccezione, fornendo come ritorno un Fault SOAP al posto della normale risposta. Nel corpo dei dettagli presenta lo stack trace della chiamata, mentre faultcode contiene il codice d'errore, come mostrato nell'esempio 9.16.

Esempio 9.16 – *Fault SOAP.*

```
<?xml version='1.0' encoding='UTF-8'?>
<soap:Envelope xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>

<soap:Body>
  <soap:Fault>
    <faultcode>soap:Server</faultcode>
    <faultstring/>
    <detail>
      <e:electric-detail xmlns:e='http://www.themindelectric.com/'>
        <class>java.lang.NullPointerException</class>
```

```
        <message/>
        <trace>
  java.lang.NullPointerException
  at net.xmethods.services.currencyexchange.CurrencyExchange.getRate(CurrencyExchange.java:196)
  at net.xmethods.services.currencyexchange.CurrencyExchange.getRate(CurrencyExchange.java:19)
  at java.lang.reflect.Method.invoke(Native Method)
  at electric.util.Function.execute(Function.java:138)
…
  at electric.net.tcp.Request.run(TCPServer.java:262)
  at electric.util.ThreadPool.run(ThreadPool.java:105)
  at java.lang.Thread.run(Thread.java:479)
        </trace>
      </e:electric-detail>
    </detail>
  </soap:Fault>
</soap:Body>
</soap:Envelope>
```

Il client visto nei paragrafi precedenti era molto semplice e non prevedeva una gestione di eventuali Fault SOAP. In applicazioni reali è però bene sempre considerare il fatto che un servizio SOAP può rispondere con un errore al posto della struttura di risposta prevista: per gestire i fault, SAAJ dispone delle classi SOAPFault, Detail e DetailEntry. La prima definisce l'intero corpo dell'errore, da cui è possibile estrarre il codice d'errore, l'attore e la stringa di descrizione utilizzando gli opportuni metodi, come getFaultCode(); le altre consentono di estrarre il contenuto del tag Detail, che può a sua volta contenere diversi tag XML.

L'esempio 9.17 mostra l'utilizzo di queste classi, un estratto di una versione migliorata — completa di gestione dell'errore — di ExchangeClient.

Esempio 9.17 – *Gestione di un Fault SOAP.*

```
protected void processFault( SOAPFault fault ) {
System.out.println("FAULT: " + fault.getFaultCode() );

  Detail d = fault.getDetail();
  Iterator ii = d.getDetailEntries();
  while( ii.hasNext() ) {
    DetailEntry de = (DetailEntry)ii.next();
    //…
  }
}
```

Il codice completo è contenuto nel file ExchangeClientFault.java. Per provarlo, utilizzare il comando:

```
Ant runClientFault
```

Un esempio "divertente"

Per completare la trattazione di JAXM, si vedrà un esempio particolare. Dilbert, protagonista dell'omonima e famosa striscia di fumetti, è un ingegnere che lavora in una non meglio specificata grande azienda statunitense. Ogni giorno Dilbert ha a che fare con capi incompetenti, colleghi fannulloni e situazioni surreali. A parte le gustose scenette di vita vissuta in cui molti potrebbero riconoscersi, Dilbert ha anche un sito Internet (http://www.dilbert.com), dove è possibile leggere la vignetta del giorno, quelle del mese passato e persino comprare libri e gadget che lo vedono come protagonista. Quello che è più interessante, però, è che il sito di Dilbert ospita un web service che fornisce online la vignetta del giorno. Ancora più interessante è che questa è codificata in base64 (viene proprio trasmessa l'immagine in formato GIF, non un link web) e che il servizio SOAP richiede l'utilizzo di alcune chiamate JAXM non immediate, come l'utilizzo di una SOAPAction particolare. Si vedrà dunque un esempio di client per accedere a questo particolare servizio.

Le cose importanti sono sostanzialmente due. Per prima cosa, è necessario impostare l'azione SOAP a http://tempuri.org/DailyDilbertImage. Questo è fattibile ottenendo dal messaggio SOAP le intestazioni MIME e impostandone una (chiamata SOAPAction) con il valore richiesto. Importante, al termine delle operazioni, è la chiamata al metodo saveChanges() sul messaggio SOAP, per memorizzare le informazioni impostate (esempio 9.18).

Esempio 9.18 – *Modifica delle intestazioni MIME per includere il SOAPAction.*

```
MimeHeaders mh = msg.getMimeHeaders();
mh.setHeader( "SOAPAction", "http://tempuri.org/DailyDilbertImage" );
msg.saveChanges();
```



In fase di ricezione della risposta è necessario utilizzare la classe MimeUtility presente in JavaMail. Questa contiene il metodo decode() che si occupa di decodificare un blocco dati codificato (con varie codifiche, tra cui base64) fornendo il risultato come stream. Una volta ottenuto lo stream decodificato, è possibile impostare un ciclo di lettura per caricare le informazioni in un array di byte tramite la classe ByteArrayOutputStream e in seguito convertirla in immagine tramite la classe ImageIcon di Swing.

Nell'esempio 9.19 è presente l'estratto di codice che esegue questa serie di operazioni.

Esempio 9.19 – *Decodifica di una immagine codificata in base64 e visualizzazione.*

```
InputStream is = MimeUtility.decode(
new ByteArrayInputStream( e.getValue().getBytes() ),
"base64"
);
```

```
ByteArrayOutputStream imageFile = new ByteArrayOutputStream();
byte[] buf = new byte[4096];
while( true ) {
int size = is.read( buf );
if( size == -1 )
break;
imageFile.write( buf, 0, size );
}

label.setText("");
label.setIcon( new ImageIcon( imageFile.toByteArray() ) );
```

Il codice completo è riportato nell'esempio 9.20.

Esempio 9.20 – *Il codice completo di DilbertClient.java.*

```
import java.awt.event.*;
import java.net.*;
import java.io.*;
import java.util.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.xml.messaging.*;
import javax.xml.soap.*;

import javax.activation.*;
import javax.naming.*;

import javax.mail.internet.*;

public class DilbertClient {

  private SOAPConnection con;
  private JLabel    label;
  private JFrame      frame;

  public DilbertClient() {
    createUI();
    createSOAPConnection();
    call();
  }
```

```
private void createUI() {
  frame = new JFrame("DilbertClient");
  label = new JLabel("Welcome to DilberClient (Version 0.1)");
  label.setBorder( new EmptyBorder( 10, 5, 10, 10 ) );
  frame.addWindowListener( new WindowAdapter() {
    public void windowClosing( WindowEvent e ) {
      System.exit(0);
    }
  });
  frame.getContentPane().add( label );
  frame.pack();
  frame.setVisible( true );
}

private void createSOAPConnection() {
  try {
  SOAPConnectionFactory scf = SOAPConnectionFactory.newInstance();
    con = scf.createConnection();
  } catch(Exception e) {
    System.err.println("Unable to open a SOAPConnection");
    e.printStackTrace();
    label.setText("Error (" + e.getMessage() + ")" );
  }
}

private void call() {

  try {
    System.out.println("DilbertClient.call(): creating SOAP message");
    label.setText("Creating data");

    MessageFactory mf   = MessageFactory.newInstance();
    SOAPMessage msg = mf.createMessage();
    SOAPPart    sp  = msg.getSOAPPart();
    SOAPEnvelope  env = sp.getEnvelope();
    SOAPHeader    hdr = env.getHeader();
    SOAPBody    bdy = env.getBody();

    SOAPBodyElement gltp = bdy.addBodyElement(env.createName("DailyDilbertImage"));

    gltp.addChildElement(
      env.createName("DailyDilbertImageSoapIn",
      "tns",
      "http://tempuri.org/")
```

```
).addTextNode("Synonym");

MimeHeaders mh = msg.getMimeHeaders();
mh.setHeader( "SOAPAction", "http://tempuri.org/DailyDilbertImage" );
msg.saveChanges();

URLEndpoint urlEndpoint
     = new URLEndpoint("http://www.esynaps.com/WebServices/DailyDiblert.asmx");

FileOutputStream sentFile = new FileOutputStream("sent.xml");
sentFile.write( "<?xml version=\"1.0\"?>".getBytes() );
msg.writeTo(sentFile);
sentFile.close();

// Send the message to the provider using the connection.
System.out.println("DilbertClient.call(): sending...");
label.setText("Sending request...");

SOAPMessage reply = con.call(msg, urlEndpoint);

if (reply != null) {

  System.out.println("DilbertClient.call(): response received");
  label.setText("Computing response");

  sp  = reply.getSOAPPart();
  env = sp.getEnvelope();
  bdy = env.getBody();

  //Ottiene DailyDilbertImageResponse
  Iterator childs = bdy.getChildElements();
  SOAPElement e = (SOAPElement)childs.next();

  //Ottiene il nodo DailyDilbertImageResult
  childs = e.getChildElements();
  e = (SOAPElement)childs.next();

  label.setText("Translating data");
  InputStream is = MimeUtility.decode(new ByteArrayInputStream( e.getValue().getBytes() ),
                                                                  "base64" );
  ByteArrayOutputStream imageFile = new ByteArrayOutputStream();
  byte[] buf = new byte[4096];
  while( true ) {
```

```
          int size = is.read( buf );
          if( size == -1 )
            break;
          imageFile.write( buf, 0, size );
        }

        label.setText("");
        label.setIcon( new ImageIcon( imageFile.toByteArray() ) );
        frame.pack();

        FileOutputStream replyFile = new FileOutputStream("reply.xml");
        replyFile.write( "<?xml version=\"1.0\"?>".getBytes() );
        reply.writeTo(replyFile);
        replyFile.close();

      } else {
        System.err.println("No reply");
        label.setText("The server is not responding");
      }

    } catch(Throwable e) {
      label.setText("Error (" + e.getMessage() + ")");
      e.printStackTrace();
    }
  }

  public static void main(String args[]) {
    new DilbertClient();
  }

}
```

Anche per DilbertClient.java è presente nel codice del capitolo il relativo file di compilazione ed esecuzione per Ant. In questo caso è solo necessario modificare il file per indicare il percorso di installazione del JWSDP e lanciare il comando Ant senza parametri.

Descrivere: WSDL e JAX-RPC

Un altro elemento fondamentale nell'insieme di tecnologie che supportano i web service è lo standard WSDL (Web Service Definition Language), che consente di descrivere un servizio in tutti i suoi aspetti. In un documento WSDL si trovano tutte le chiamate che è possibile fare a un web service, le specifiche delle strutture dati di input e output, gli URL per accedere ai servizi.

Inoltre WSDL non è uno standard legato a un livello di trasporto particolare (come SOAP), ma è aperto all'utilizzo con protocolli differenti, specificando di volta in volta *bindings* ("collegamenti") a questo o a quell'altro protocollo. Nonostante ciò, WSDL privilegia SOAP, indicando in una sezione delle sue specifiche le modalità di collegamento a questo standard. Con in mano un documento WSDL, un integratore di sistemi è in grado di sapere come dialoga un determinato servizio, oppure un sistema software evoluto può dinamicamente invocare un web service ed elaborarne il risultato. Questo tipo di informazioni sono quelle che dovranno essere presenti all'interno dei registri di web service: costituiscono le specifiche tecniche per l'accesso a un servizio.

Perché WSDL

L'utilizzo di WSDL ha due vantaggi: per prima cosa, alcuni strumenti di sviluppo sono in grado di prendere in input un file WSDL e produrre del codice che implementa l'infrastruttura per realizzare il servizio, oppure per crearne un runtime per la semplice implementazione di un client; in secondo luogo, è un modo per disaccoppiare il web service dal protocollo di trasporto e dai percorsi fisici, come gli endpoint.

Come accennato, nel file WSDL è presente l'indicazione del protocollo da utilizzare: un sistema di runtime che supporti diversi protocolli di comunicazione XML, come SOAP ed XML-RPC, potrebbe capire dal WSDL come ricondurre a messaggi SOAP o XML-RPC reali le rappresentazioni astratte contenute nel documento WSDL. Un servizio potrebbe infatti partire utilizzando SOAP come protocollo, ma poi passare a XML-RPC con l'intento di ottimizzarne le prestazioni, ad esempio per via della sua potenziale minor occupazione di banda. Se il client del servizio viene codificato direttamente su SOAP, il suo passaggio a XML-RPC potrebbe essere problematico. Con un runtime dinamico che supporti entrambi i protocolli e che faccia riferimento al file WSDL per conoscere quale protocollo deve essere fisicamente utilizzato, la migrazione potrebbe essere più indolore.

Sicuramente uno scenario del genere comporta l'utilizzo di infrastrutture per i web service più complesse di quelle fornite da JAXM e SAAJ. Inoltre, un'esigenza di questo tipo potrebbe più facilmente nascere in un futuro, magari quando esisterà un legacy di web service in SOAP e XML-RPC e si dovrà migrare a un ipotetico nuovo protocollo di comunicazione XML.

Ma anche senza considerare aspetti così remoti, l'utilizzo dinamico di WSDL potrebbe limitarsi anche alla sola acquisizione dell'endpoint del servizio: invece che configurare direttamente l'URL del servizio, questo andrebbe letto dal documento WSDL. Si introduce così un livello di indirezione, simile a quanto fanno i servizi di DNS sugli indirizzi Internet.

Il vantaggio di avere un livello di indirezione è che, se si hanno molti client che puntano a un file WSDL per ottenere l'endpoint di un servizio, nel momento in cui questo cambia indirizzo non è necessario riconfigurare tutti i client, ma è sufficiente modificare il solo file WSDL.

All'interno di WSDL

Un documento WSDL è un file XML contentente un insieme di definizioni. Nello standard WSDL è possibile trovare quattro diversi tipi di informazioni. Sicuramente l'aspetto principale

sono l'insieme di regole che consentono di definire in modo astratto un web service; oltre a ciò, sono presenti le specifiche per collegare il servizio a SOAP, HTTP e MIME.

La definizione astratta dei web service di WSDL è composta da cinque diversi elementi:

- i tipi (*types*);
- i messaggi (*messages*);
- le operazioni (*portType*);
- i collegamenti (*bindings*);
- la definizione del servizio (*service*);

La struttura di un file WSDL è presente nell'esempio 9.21.

Esempio 9.21 – *Struttura di un file WSDL.*

```
<wsdl:definitions name="nmtoken"? targetNamespace="uri"?>

  <import namespace="uri" location="uri"/>*

  <wsdl:documentation .... /> ?

  <wsdl:types> ?
    <wsdl:documentation .... />?
    <xsd:schema .... />*
    <-- extensibility element --> *
  </wsdl:types>

  <wsdl:message name="nmtoken"> *
    <wsdl:documentation .... />?
    <part name="nmtoken" element="qname"? type="qname"?/> *
  </wsdl:message>

  <wsdl:portType name="nmtoken">*
    <wsdl:documentation .... />?
    <wsdl:operation name="nmtoken">*
      <wsdl:documentation .... /> ?
      <wsdl:input name="nmtoken"? message="qname">?
        <wsdl:documentation .... /> ?
      </wsdl:input>
```

```
      <wsdl:output name="nmtoken"? message="qname">?
        <wsdl:documentation .... /> ?
      </wsdl:output>
      <wsdl:fault name="nmtoken" message="qname"> *
        <wsdl:documentation .... /> ?
      </wsdl:fault>
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="nmtoken" type="qname">*
    <wsdl:documentation .... />?
    <-- extensibility element --> *
    <wsdl:operation name="nmtoken">*
      <wsdl:documentation .... /> ?
      <-- extensibility element --> *
      <wsdl:input> ?
        <wsdl:documentation .... /> ?
        <-- extensibility element -->
      </wsdl:input>
      <wsdl:output> ?
        <wsdl:documentation .... /> ?
        <-- extensibility element --> *
      </wsdl:output>
      <wsdl:fault name="nmtoken"> *
        <wsdl:documentation .... /> ?
        <-- extensibility element --> *
      </wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="nmtoken"> *
    <wsdl:documentation .... />?
    <wsdl:port name="nmtoken" binding="qname"> *
      <wsdl:documentation .... /> ?
      <-- extensibility element -->
    </wsdl:port>
    <-- extensibility element -->
  </wsdl:service>

  <-- extensibility element --> *

</wsdl:definitions>
```

Tipi

I tipi definiti da WSDL sono l'equivalente delle strutture (*struct*) del linguaggio C: strutture dati anche complesse che sono utilizzate come elementi base per costruire i messaggi di input e output definiti nella sezione "messaggi". Ad esempio, la seguente porzione di file XML (esempio 9.22), definisce due elementi: TradePriceRequest, composto da una stringa e TradePrice, composto da un valore in virgola mobile.

Esempio 9.22 – *Un esempio di definizione di tipi.*

```
<types>
  <schema targetNamespace="http://example.com/stockquote.xsd"
    xmlns="http://www.w3.org/2000/10/XMLSchema">
    <element name="TradePriceRequest">
      <complexType>
        <all>
          <element name="tickerSymbol" type="string"/>
        </all>
      </complexType>
    </element>
    <element name="TradePrice">
      <complexType>
        <all>
          <element name="price" type="float"/>
        </all>
      </complexType>
    </element>
  </schema>
</types>
```

In questo caso, dove la struttura dati contiene un solo campo, la definizione di un tipo specifico non è indispensabile: la sua utilità è per lo più relativa alla manutenibilità del documento WSDL. In questo modo infatti viene creato un "alias" e nel resto del documento WSDL si utilizzeranno i tipi definiti al posto dei tipi di dati primitivi, disaccoppiando il resto delle definizioni dai tipi di dati reali.

Messaggi

I messaggi sono gli elementi che costituiscono gli input e gli output dei servizi. I singoli messaggi possono contenere i tipi di dati complessi definiti nella sezione types (fig. 9.3), oppure semplici dati primitivi. Un messaggio è presente nell'esempio 9.23, ed è relativo al file WSDL del servizio di cambio valute utilizzato precedentemente. Come si può notare osservando il listato, vengono definiti due diversi messaggi, uno relativo alla richiesta, getRateRequest, e uno relativo alla risposta, getRateResponse. Il primo definisce due elementi, di tipo stringa, chiamati country1 e country2; il secondo definisce un unico elemento di risposta Result, di tipo float.

Figura 9.3 – *La struttura di un messaggio WSDL.*

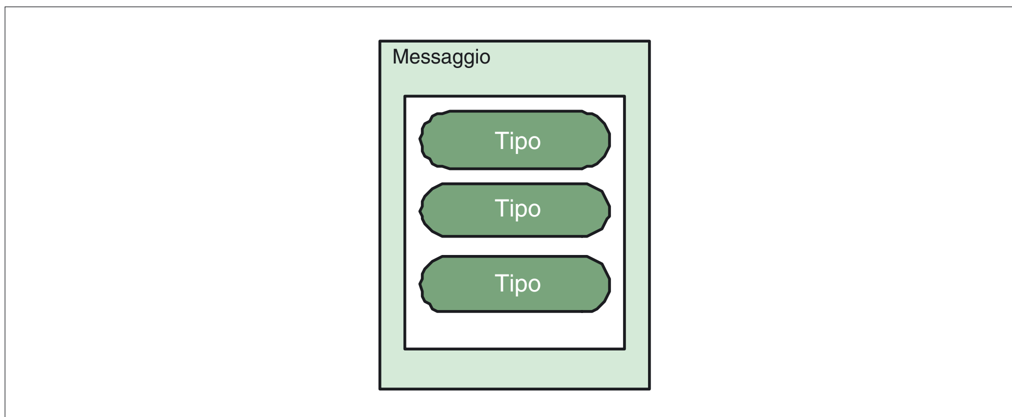
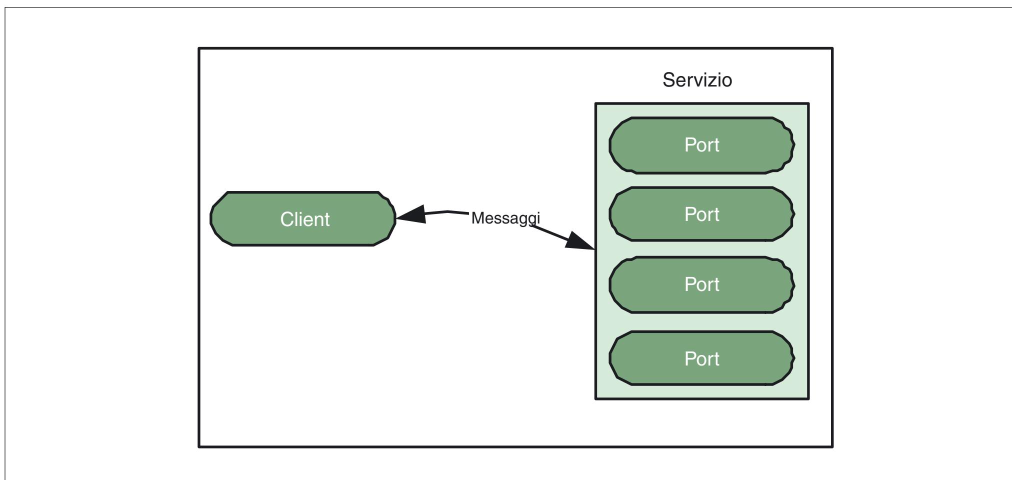


Figura 9.4 – *L'accesso al web service avviene tramite le "port".*



Esempio 9.23 – *Un esempio di definizione di messaggi.*

```
<message name="getRateRequest">
  <part name="country1" type="xsd:string"/>
  <part name="country2" type="xsd:string"/>
</message>
<message name="getRateResponse">
```

```
<part name="Result" type="xsd:float"/>
</message>
```

Operazioni

Questa sezione definisce le operazioni fornite dal web service. Se si ponessero in relazione i web service con la programmazione distribuita, ad esempio RMI, le operazioni WSDL equivarrebbero ai singoli metodi dell'oggetto remoto (fig. 9.4). Per ciascuna operazione, WSDL definisce i messaggi di input e di output. Un'operazione di riferimento è presente nell'esempio 9.24, sempre estratto dal WSDL del servizio di cambio valuta.

Esempio 9.24 – *Un'operazione di esempio.*

```
<portType name="CurrencyExchangePortType">

  <operation name="getRate">
    <input message="tns:getRateRequest" />
    <output message="tns:getRateResponse" />
  </operation>
</portType>
```

L'elemento <operation> definisce una singola operazione e può contenere opzionalmente almeno uno dei due sottoelementi <input> ed <output>. La presenza e l'ordine di questi elementi determina la tipologia di servizio, che può rientrare all'interno di quattro diverse nature:

- *One-way*. Detto anche detto *fire&forget*, è una configurazione dove l'endpoint si limita a ricevere il messaggio inviato dal client. In questo caso è presente un solo elemento di input.

- R*equest-response.* In questo caso l'endpoint riceve un messaggio di richiesta, esegue l'elaborazione necessaria e restituisce al client un messaggio di risposta correlato alla richiesta ricevuta. Sono presenti gli elementi input e output.

- *Solicit-response.* È l'opposto del precedente. In questo caso è l'endpoint che inizia la comunicazione inviando un messaggio al client che a sua volta dovrà rispondere di conseguenza. Nel file WSDL è presente prima l'elemento output e poi quello input.

- *Notification*. È l'opposto della tipologia one-way. In questo caso è l'endpoint a inviare un messaggio al client, senza che questo debba inviare una risposta. È presente il solo elemento output.

Il modello di interazione da utilizzare dipende dalla natura del servizio. Se l'interazione *request-response* potrebbe essere quella usata per la maggior parte dei servizi, in quanto costituisce il

tipico modello di comunicazione RPC, altre modalità possono essere utili in altri ambiti. Ad esempio, una interazione *one-way* può essere utile in sistemi dove il client invia una serie di informazioni al servizio, e questo si limita a raccoglierle. In questo caso il client non è interessato al risultato dell'elaborazione dei suoi dati, come nell'esempio relativo alla raccolta dati ambientali descritto all'inzio del capitolo.

Collegamenti

In questa sezione avviene la mappatura del servizio astratto, definito nelle sezioni precedenti, al protocollo concreto di comunicazione (ad esempio SOAP). Il contenuto di questa sezione è fortemente dipendente dal protocollo utilizzato, e quindi dalle estensioni WSDL relative.

In WSDL, la struttura di un collegamento (semplificata) è quella riportata nell'esempio 9.25. Quella completa è presente nella struttura completa di un file WSL riportata a inizio paragrafo.

Esempio 9.25 – *Sezione WSDL relativa ai collegamenti.*

```
<wsdl:binding name="nmtoken" type="qname">*
<wsdl:operation name="nmtoken">*
<wsdl:input> ?
</wsdl:input>

    <wsdl:output> ?
</wsdl:output>

    <wsdl:fault name="nmtoken"> *
</wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>
```

Come si può notare, un collegamento può contenere diverse operazioni, ciascuna delle quali può avere un elemento di input, di output e una serie di elementi di errore (fault). Nell'esempio 9.26 è presente un esempio di collegamento con SOAP, sempre estratto dal file WSDL per il servizio di cambio valuta. La struttura del collegamento è la seguente: per prima cosa, viene definito lo stile del collegamento (può essere `rpc` oppure `document`) e il tipo di trasporto:

```
<soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
```

In questo caso la comunicazione è di tipo RPC e il trasporto avviene sul protocollo HTTP. In secondo luogo, è necessario definire, per ciascuna operazione, come trattare i messaggi di input ed output. Essendo questo un servizio request-response, le informazioni sono presenti entrambe. Il messaggi di input è così definito:

```
<soap:body use="encoded" namespace="urn:xmethods-CurrencyExchange" encodingStyle
="http://schemas.xmlsoap.org/soap/encoding/"/>
```

In questo elemento viene indicato il namespace del messaggio (urn:xmethods-CurrencyExchange) e il tipo di encoding. L'attributo use può assumere i valori econding, oppure literal. Nel primo caso, l'attributo encodingStyle dovrà indicare il tipi di encoding utilizzato. In questo caso, viene utilizzato l'encoding SOAP.

Esempio 9.26 – *Un esempio di collegamento.*

```
<binding name="CurrencyExchangeBinding" type="tns:CurrencyExchangePortType">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getRate">
      <soap:operation soapAction=""/>

      <input >
        <soap:body use="encoded" namespace="urn:xmethods-CurrencyExchange" encodingStyle
                              ="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output >
        <soap:body use="encoded" namespace="urn:xmethods-CurrencyExchange" encodingStyle
                              ="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>
  </binding>
```

Definizione del servizio

L'ultimo elemento di un file WSDL è la definizione del servizio: questa sezione consente di raccogliere tutte le operazioni sotto un unico nome (fig. 9.4). Il servizio è identificato da un nome (l'attributo name dell'elemento service) e può avere una descrizione, contenuta nel sottolemento opzionale documentation. All'interno dell'elemento service vengono elencate tutte le operazioni esposte dal servizio, sotto forma di elementi port. Per ciascuno di questi viene indicato il collegamento utilizzato. Il contenuto dell'elemento port cambia in funzione del tipo di collegamento utilizzato. Nell'esempio 9.27 è presente una definizione di servizio, in particolare del servizio di cambio valute, dove il collegamento utilizzato è quello con SOAP. In questo caso nell'elemento port è presente un elemento address appartenente al namespace soap: (utilizzato in tutto il WSDL per riferirsi agli elementi strettamente relativi al collegamento di WSDL con SOAP). Questo elemento indica l'URL fisico dell'endpoint del servizio.

Esempio 9.27 – *Definizione di servizio.*

```
<service name="CurrencyExchangeService">

    <documentation>Returns the exchange rate between the two currencies</documentation>
    <port name="CurrencyExchangePort" binding="tns:CurrencyExchangeBinding">
      <soap:address location="http://services.xmethods.net:80/soap"/>
```

```
    </port>
  </service>
```

L'utilizzo di WSDL può diventare anche molto complesso nel momento in cui ci si addentri in collegamenti e protocolli diversi da SOAP e HTTP. Per ciascun protocollo e livello di trasporto è infatti necessario definire una estensione WSDL specifica e costruire i documenti WSDL secondo questa grammatica. Il vantaggio della grande estensibilità di WSDL si paga quindi in termini di complessità.

JAX-RPC

All'inizio del paragrafo "Perché WSDL", si è fatto cenno all'utilità di questo standard di descrizione dei web service quando associato a strumenti di generazione di codice volto a supportare gli specifici servizi descritti nei documenti WSDL. JAX-RPC (Java API for XML Remote Procedure Call) contiene al suo interno proprio questa funzionalità, facendo leva su WSDL per semplificare l'accesso e la produzione di web service. Piuttosto che una API, infatti, JAX-RPC è un insieme di regole e strumenti per nascondere allo sviluppatore i dettagli della comunicazione SOAP tra i diversi nodi.

Se da una parte JAXM e SAAJ forniscono un controllo fine sul messaggio SOAP, con la possibilità di intervenire su ogni aspetto di ciascun elemento come il namespace e gli attributi, dall'altra questo aspetto comporta la necessità di scrivere molto codice per implementare ciascun elemento del messaggio, come la busta (envelope), l'intestazione (header) e il corpo del messaggio (body).

In contrapposizione, o meglio, a complemento di JAXM e SAAJ, è nata JAX-RPC. Questa tecnologia sta a JAXM come RMI sta alla comunicazione tramite socket: semplifica la realizzazione di applicazioni distribuite basate su protocolli di comunicazione.

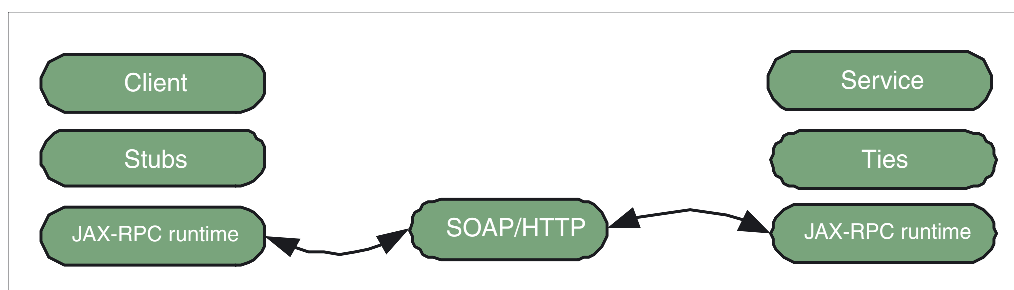
Potrebbe sembrare che JAXM e JAX-RPC si sovrappongano, in quanto entrambe implementano SOAP. In realtà, JAXM consente semplicemente di creare, consumare e inviare messaggi SOAP, mentre JAX-RPC si occupa degli aspetti ad alto livello, e cioè di mappare interfacce e classi Java rendendole disponibili come servizi web. Inoltre non esiste un vincolo a SOAP: a oggi è l'unico protocollo supportato, ma potrebbero aggiungersene di nuovi in futuro. In realtà, JAXM e SAAJ possono fornire a JAX-RPC l'implementazione del protocollo SOAP, ponendosi quindi come tecnologia sottostante e complementare alle API per l'RPC in XML. JAX-RPC 1.0 supporta SOAP 1.1, WSDL 1.1, SOAP con allegati e le estensioni MIME.

Architettura di JAX-RPC

JAX-RPC si basa su una architettura composta da codice intermedio che implementa la comunicazione a basso livello, lasciando a livello del programmatore solo le interfacce del servizio. In fig. 9.5 è presente un esempio, dove sia il client che il server utilizzano JAX-RPC per la comunicazione.

Il client dialoga con una interfaccia, generata da strumenti di sviluppo (in JWSDP è presente lo strumento `xrpcc`), prodotta a partire da un file WSDL. L'interfaccia è implementata da una classe che implementa i dettagli della comunicazione SOAP, lo *stub*. Al di sotto di questo, il runtime di

Figura 9.5 – *Architettura di JAX-RPC.*



JAX-RPC si occupa della comunicazione tramite SOAP/HTTP con il *tie*, il corrispondente dello stub sul lato server. È poi compito del tie di comunicare con la reale implementazione del servizio.

È però possibile utilizzare `xrpcc` per generare solo uno degli strati: quello client o quello server. Per creare uno stub è sufficiente avere a disposizione un file WSDL: dato in input allo strumento `xrpcc`, produce tutto il codice necessario per comunicare con il servizio. Il client non deve far altro che importare il package relativo al codice generato e deciso prima dallo sviluppatore e istanziare la classe che rappresenta il servizio.

Per generare il lato server, è possibile partire, oltre che dal documento WSDL, anche da una interfaccia Java che estenda `java.rmi.Remote` e dove ciascun metodo sollevi l'eccezione `java.rmi.RemoteException`. A partire da questa, `xrpcc` è in grado di generare i tie e il documento WSDL di descrizione, oltre agli altri file necessari per installare il servizio in un web container.

Mappatura dei tipi

Nell'implementare la corrispondenza biunivoca tra WSDL e Java, JAX-RPC deve basarsi su un preciso standard che indichi in che modo un determinato tipo, presente in uno dei due mondi, debba essere rappresentato nell'altro. JAX-RPC dispone di una mappatura dei dati primitivi (tab. 9.3), ma anche degli array, delle strutture dati complesse e delle enumerazioni.

Se da una parte i dati primitivi trovano spesso diretta corrispondenza, e gli array WSDL possono essere mappati direttamente in array Java, le strutture dati complesse e le enumerazioni WSDL richiedono più lavoro. In questi casi, classi apposite vengono generate dagli strumenti di sviluppo. Nel primo caso vengono generati componenti JavaBeans strutturati secondo il modello presente nel documento WSDL.

Modelli di interazione

Nonostante WSDL definisca quattro diversi modelli di interazione tra il client e il servizio, JAX-RPC supporta solo i modelli one-way e request-response, più uno diverso, detto *invocazione non-bloccante*. Quest'ultimo è una variante della modalità request-response e prevede l'invio di un messaggio di richiesta e la ricezione di un messaggio di risposta, ma nel tempo che intercorre tra la ricezione della richiesta e la produzione della risposta, il client è svincolato dall'attesa.

Questa modalità è disponibile però solo all'interno di container J2EE, dove è il server a poter rimanere in attesa della risposta al posto del client.

Un client JAX-RPC

Per creare un programma client con JAX-RPC è necessario partire dal file WSDL di definizione del servizio e utilizzare lo strumento xrpcc per creare il codice di supporto, ad esempio con il comando seguente:

```
xrpcc.bat -d classes -s src -client -keep config.xml
```

Lo strumento supporta molte opzioni, in questo caso gli è stato indicato di posizionare le classi generate nella cartella classes, i sorgenti nella cartella src e di generare i soli stub (opzione -client). Inoltre, con l'opzione -keep, non vengono cancellati i file intermedi, come i sorgenti del codice di infrastruttura generato. L'ultimo parametro del comando è il file di configurazione di xrpcc che indica cosa generare e contiene il riferimento al file WSDL da analizzare. Lo strumento xrpcc e il formato del file di configurazione non sono parte delle specifiche JAX-RPC e dunque possono essere diversi in implementazioni diverse di questo standard. Il contenuto del file config.xml è presente nell'esempio 9.28.

Tabella 9.3 – *Mappatura dati primitivi tra WSDL e Java.*

WSDL	Java
xsd:string	java.lang.String
xsd:integer	java.math.BigInteger
xsd:int	int
xsd:long	long
xsd:short	short
xsd:decimal	java.math.BigDecimal
xsd:float	float
xsd:double	double
xsd:boolean	boolean
xsd:byte	byte
xsd:QName	javax.xml.rpc.namespace.QName
xsd:dateTime	java.util.Calendar
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]

Esempio 9.28 – *File di configurazione config.xml per xrpcc.*

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <wsdl location="
  file:///d:/max/codice/jax-rpc/client/CurrencyExchangeService.xml"
  packageName="currencyExchange"/>
</configuration>
```

Gli elementi interessanti del file di configurazione sono l'attributo location che indica l'URL del WSDL sorgente (il percorso dovrà riflettere la reale posizione del file su disco) e il nome del package in cui posizionare il codice. Lo strumento xrpcc, infatti, a fronte di un documento WSDL anche semplice, genera molte classi, ed è dunque buona norma racchiuderle in un package dedicato.

In questo caso è stato generato lo stub per l'accesso al servizio di cambio valute di XMethods. Analizzando il codice generato, si può notare la presenza dell'interfacca CurrencyExchangeService che rappresenta il web service contenuto nel file WSDL. L'implementazione concreta di questa interfaccia è data dalla classe CurrencyExchangeService_Impl, che può essere istanziata direttamente. L'interfaccia dispone di un sigolo metodo per ottenere lo stub dell'unica operazione disponibile nel servizio, quella di cambio valute. Il metodo getCurrencyExchangePort() restituisce infatti un oggetto di tipo CurrencyExchangePortType che a sua volta è dotato del metodo getRate(String, String), rappresentazione Java della singola operazione effettuabile da questo servizio web.

Nell'esempio 9.29 è presente il codice completo del client. Come si può notare, è molto più compatto dell'equivalente JAXM.

Esempio 9.29 – *ExchangeClient2.java*

```
import java.rmi.*;
import javax.xml.rpc.*;

import currencyExchange.*;

class ExchangeClient2 {

  public ExchangeClient2() {
    sendMessage( "us", "euro" );
  }

  protected void sendMessage( String divisaIniziale, String divisaFinale ) {
    try {
      CurrencyExchangeService_Impl service = new CurrencyExchangeService_Impl();
      CurrencyExchangePortType portType = service.getCurrencyExchangePort();

      String endpoint = System.getProperty("endpoint");
```

```
            if( endpoint != null ) {
               ((Stub)portType)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY, endpoint);
            }

            System.out.println( portType.getRate( divisaIniziale, divisaFinale ) );
         } catch( RemoteException re ) {
            re.printStackTrace();
         }
      }

      public static void main( String[] args ) {
         new ExchangeClient2();
      }
   }
```

Si osservi la gestione dell'endpoint: l'indirizzo fisico del servizio viene inserito direttamente nel codice generato da parte dello strumento xrpcc, in quanto presente nel file WSDL. Potrebbe essere necessario però puntare allo stesso servizio in posizioni differenti. Per fare ciò è necessario impostare la proprietà ENDPOINT_ADDRESS_PROPERTY dello stub.

In ExchangeClient2.java, l'endpoint viene prelevato da una proprietà di sistema:

```
String endpoint = System.getProperty("endpoint");
```

Se questa non è nulla (la proprietà è stata effettivamente passata alla Virtual Machine), viene passata all'oggetto port tramite il metodo _setProperty():

```
if( endpoint != null ) {

((Stub)portType)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY, endpoint);
}
```

L'oggetto port può essere convertito in un oggetto di tipo stub, in quanto tutti gli oggetti di questo tipo implementano questa interfaccia.

Anche per questo esempio è presente nel codice del capitolo il relativo file di compilazione ed esecuzione per Ant. In questo caso è solo necessario modificare il file per indicare il percorso di installazione del JWSDP e lanciare il comando Ant senza parametri. L'esempio verrà compilato ed eseguito. Il file di build contiene inoltre la proprietà endpoint per puntare alla servlet JAXM che implementa questo servizio. Se non viene indicata nessuna proprietà nel file di build, il client accede al servizio di XMethods su Internet.

Realizzare un servizio

Per creare un web service con JAX-RPC è possibile partire, oltre che dal WSDL, anche da una interfaccia che implementi java.rmi.Remote. In questo esempio viene replicato il servizio di cambio valute, utilizzando però, come punto di partenza un'interfaccia Java (esempio 9.30).

Esempio 9.30 – *CuurrencyExchange.java.*

```
package galacticExchange;

import java.rmi.*;

public interface CurrencyExchange extends Remote {
  public float getRate( String country1, String country2 ) throws RemoteException;
}
```

Come si nota dal codice, il servizio è contenuto nel package galacticExchange: questo è infatti un servizio di fantasia, che ha la medesima struttura di quello di XMethods, ma che vorrebbe trattare di valute "intergalattiche". L'implementazione completa è presente nell'esempio 9.31: come si nota non ci sono riferimenti a XML e tantomeno a SOAP.

Esempio 9.31 – *CurrencyExchangeImpl.java.*

```
package galacticExchange;

import java.rmi.*;

public class CurrencyExchangeImpl implements CurrencyExchange {
  public float getRate( String country1, String country2 ) throws RemoteException {
    return (float)20.10;
  }
}
```

Come si nota osservando il codice dell'implementazione del servizio, questo ritorna una semplice costante, in modo simile a quello prodotto con JAXM.

Una volta compilato il codice per installare il servizio all'interno di un web container, è necessario creare un file WAR con il codice compilato e un file XML di descrizione, jaxrpc-ri.xml. Quest'ultimo è specifico dell'implementazione di JAX-RPC del JWSDP e contiene informazioni relative ai nomi dei servizi, agli URL da assumere all'interno del container, e alle classi che implementano i servizi. Nell'esempio 9.32 è presente il contenuto del file jaxrpc-ri.xml utilizzato in questo esempio.

Esempio 9.32 – *File jaxrpc-ri.xml per il servizio di cambio valute galattiche.*

```
<?xml version="1.0" encoding="UTF-8"?>

<webServices
  xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/dd"
  version="1.0"
```

```
targetNamespaceBase="http://schemas.mokabyte.it/wsdl"
typeNamespaceBase="http://schemas.mokabyte.it/types"
urlPatternBase="/ws">

<endpoint
  name="GalacticCurrencyExchangeEndpoint"
  displayName="GalacticCurrencyExchange"
  description=""
  interface="galacticExchange.CurrencyExchange"
  implementation="galacticExchange.CurrencyExchangeImpl"/>

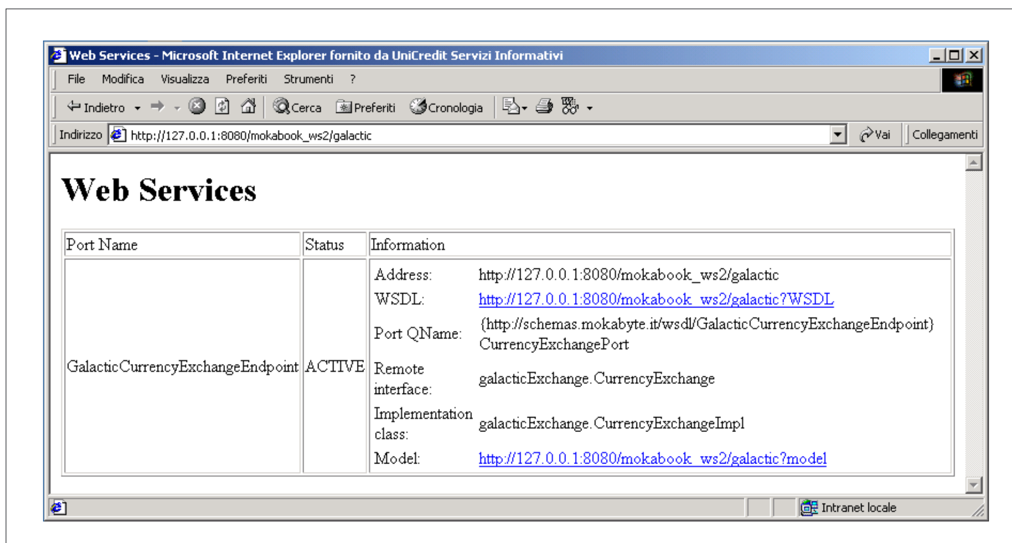
<endpointMapping
  endpointName="GalacticCurrencyExchangeEndpoint"
  urlPattern="/galactic"/>

</webServices>
```

Una volta in possesso dell'interfaccia, dell'implementazione e del file di descrizione, è possibile passare alla fase di creazione del servizio. Con JWSDP è necessario utilizzare il comando:

```
wsdeploy.bat -o <nuovo file> <vecchio file>
```

Figura 9.6 – *Visualizzazione degli endpoint del servizio GalacticExchange.*



WSDeploy si occupa di estrarre le informazioni dal WAR, creare i tie dei servizi, creare i file di supporto necessari e di ricompattare tutto all'interno di un nuovo file WAR, che potrà essere poi installato in un web container. Tutte queste operazioni possono essere fatte in automatico da Ant tramite il file di build presente nel codice del capitolo.

Una volta installato il servizio, è possibile analizzarne la struttura puntando all'indirizzo indicato nel file jaxrpc-ri.xml. In questo caso, all'URL http://127.0.0.1/mokabook_ws2/galactic è possibile accedere alla visualizzazione degli endpoint del servizio (fig. 9.6) e visualizzare il file WSDL del servizio.

Per provare il servizio è presente il programma ExchangeClient3.java: la sua struttura è identica a ExchangeClient2.java, con la sola differenza che il nome del servizio puntato è diverso (e di conseguenza diverse le interfacce Java che fungono da proxy). Il listato completo di ExchangeClient3.java è presente nel codice del capitolo.

Catalogare e scoprire: UDDI e JAXR

Come accennato all'inizio del capitolo, la visione dei web service non può essere completa senza un punto di incontro tra fornitori e potenziali clienti dei servizi. Questa funzione è demandata ai registri di web service, tra i quali UDDI (Universal Directory & Description Interface) è quello più consolidato.

Caratteristiche di UDDI

Il registro UDDI è supportato da una rete mondiale di nodi, collegati tra di loro in una sorta di federazione, in modo similare alla tecnologia DNS. Quando un client sottopone una informazione al registro, questo la propaga agli altri nodi. In questo modo si attua la ridondanza dei dati, fornendo una certa affidabilità. Il ruolo del singolo nodo rimane comunque fondamentale, poiché, nel momento in cui un client gli sottopone dei dati, questo ne diviene il *proprietario*, e sarà in futuro solo questo nodo a poter operare importanti operazioni sui dati, quali la loro eliminazione.

Le chiamate a UDDI sono implementate tramite comunicazioni SOAP di tipo request-response e permettono sostanzialmente due operazioni: quella di pubblicazione delle informazioni e quelle di ricerca delle stesse. Le informazioni gestite dal registro UDDI sono di tre tipi:

- Pagine bianche. Contengono informazioni anagrafiche delle aziende, come l'indirizzo e i numeri di telefono. Costituiscono un primo approccio alle aziende, di tipo umano prima che tecnico.

- Pagine gialle. Definiscono la categorizzazione (tassonomia) dei servizi e delle aziende. Un'azienda potrebbe essere infatti catalogata in base alla sua posizione geografica o al proprio settore industriale. Le tassonomie dovrebbero rispettare standard internazionali, quali quelli dettati dall'ISO.

- Pagine verdi. Costituiscono le informazioni tecniche dei servizi, quelle utili a livello informatico. Queste riguardano gli URL dei servizi o gli eventuali documenti WSDL.

Purtroppo a oggi non sembra che il registro UDDI contenga molte informazioni, e le poche presenti risultano nella maggioranza dei casi non valide. In attesa che le aziende superino i timori relativi allo spostare i propri servizi sul web, la piattaforma Java ha predisposto un' API opportuna per l'accesso a questo ed altri registri.

JAXR

L'accesso ai registri di web service dalla piattaforma Java avviene tramite le Java API for XML Registries (JAXR). Si parla di registri in generale perché JAXR ha un'architettura multiregistro, capace di accedere anche ad altre tipologie di registry, come ebXML R. Inoltre JAXR può propagare una singola ricerca di servizi a più registri contemporaneamente.

L'accesso ai registri avviene tramite una connessione, ottenuta dalla classe ConnectionFactory, che viene opportunamente configurata tramite una serie di proprietà che indicano, tra le altre cose, l'URL del registro al quale si vuole accedere.

Una volta in possesso della connessione è possibile eseguire ricerche e sottoporre la registrazione di aziende e servizi. Quest'ultima operazione è però sottoposta ad autenticazione e per poterla portare a termine è necessario essere registrati presso il particolare registro a cui si vuole accedere.

Un aspetto interessante di JAXR è il suo uso del pattern Futures (anche detto Lazy-Loading): una ricerca su un registro mondiale può potenzialmente restituire una grande mole di informazioni che non è garantito venga poi analizzata tutta. Per ovviare a questo inconveniente, ad ogni ricerca, gli oggetti creati da JAXR non contengono tutte le informazioni, ma solo quelle indispensabili. Nel momento in cui si renda necessario ottenere quelle mancanti, il runtime di JAXR esegue l'accesso ai dati necessari. Questa caratteristica ha due aspetti: per prima cosa, vengono recuperati solo i dati effettivamente necessari, con un guadagno in termini di performance; in secondo luogo, i client JAXR devono per forza essere sempre connessi alla rete, in modo che i successivi caricamenti dati possano essere eseguiti correttamente.

Architettura

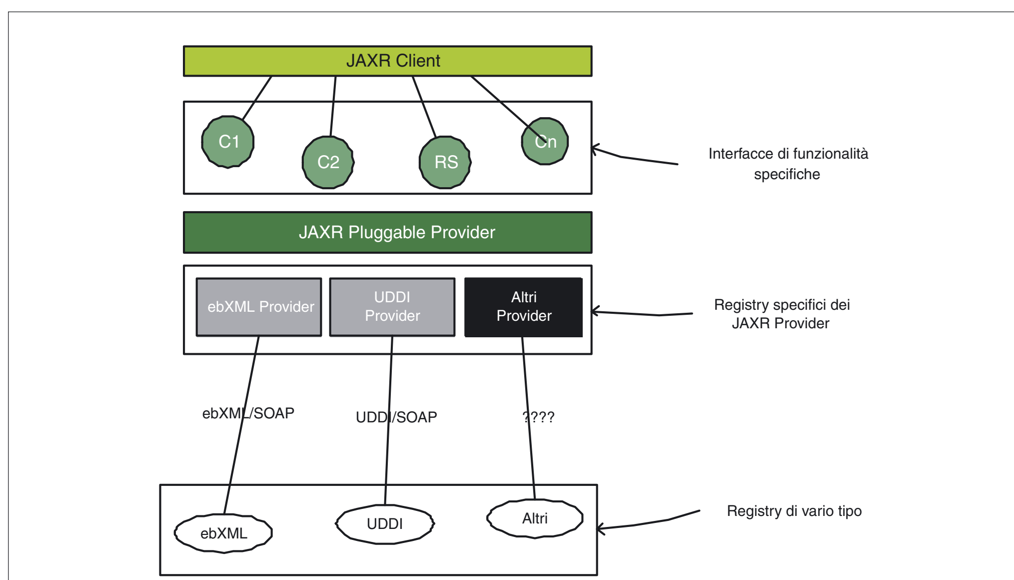
Una implementazione JAXR è detta JAXR Provider e può essere di livello 0 o livello 1. Questo indica quali elementi sono supportati dall'implementazione. Ad esempio, il supporto a UDDI è di livello 0. Questo significa che qualsiasi prodotto che dichiari la compatibilità a JAXR dovrà per forza supportare l'accesso a UDDI. Il supporto ebXML è invece posto a livello 1, quindi i prodotti JAXR level 1 compliant supporteranno sia UDDI che ebXML.

Il supporto a diversi registri di web Services avviene tramite l'utilizzo di un modello informativo comune ma estendibile che è stato mutuato dal modello informativo di ebXML con aggiunte prelevate da UDDI. Questo ha consentito di risparmiare il tempo necessario a sviluppare un modello apposito e si traduce nelle specifiche come pochissime pagine dedicate alla mappatura di JAXR su ebXML.

In fig. 9.7 è possibile vedere l'architettura generale di JAXR: un client utilizza l'interfaccia RS (RegistryService) per ottenere informazioni sulla disponibilità delle funzionalità per lo specifico prodotto, come l'indicazione del livello di supporto dell'implementazione.

Le altre interfacce (C?) supportano le varie funzionalità, come la gestione del ciclo operativo e la gestione delle query. Gli altri elementi dell'architettura sono il codice comune a tutti i registri,

Figura 9.7 – *Architettura JAXR.*



contenuto in JAXR Pluggable Provider, e i componenti di accesso agli specifici registri ebXML, UDDI o altri. Le implementazioni delle API JAXR sono racchiuse in due package:

- javax.xml.registry.infomodel che contiene le interfacce per modellare le informazioni supportate da JAXR, come pure la loro interrelazione;
- javax.xml.registry che contiene la definizione dell'interfaccia di accesso al registro.

Entità e classificazioni

Le specifiche JAXR definiscono una serie di entità che costituiscono il modello informativo su cui questo si basa. Come detto, queste sono entità presenti nel modello ebXML (anche i nomi sono sempre praticamente identici), con aggiunte mutuate da UDDI. Le principali sono:

- *Organization*. Rappresenta una azienda, ad esempio quella che ha inviato informazioni sui propri servizi al registro. Può avere un riferimento alla "capogruppo" per modellare eventuali gruppi aziendali.
- *Service*. È il servizio offerto dall'azienda.
- *ServiceBinding*. Sono le specifiche tecniche per l'accesso al servizio. Un servizio può avere più binding.

- *ClassificationScheme*. Rappresenta la tassonomia che può essere usata per classificare o categorizzare altri elementi del registro.

- *Classification*. Utilizzato per classificare concretamente un elemento del registro secondo uno specifico schema.

Attorno a queste entità ne esistono altre che consentono di collegarle tra di loro o di estendere il modello informativo, tra cui quelle per la gestione della tassonomia.

JAXR supporta due diverse tipologie di tassonomia: interna ed esterna. Queste sono relative al registro: quando una tassonomia è riconosciuta da quest'ultimo, potrà essere utilizzata come interna. In alternativa è possibile mescolare diverse tassonomie, o definirne di nuove, e creare così tassonomie esterne. Solitamente le tassonomie vengono definite da organismi nazionali o internazionali, come nel caso di NAICS (North American Industrial Classification System), standard in uso nelle nazioni dell'America del nord. Entrambe le tipologie di tassonomie hanno vantaggi e svantaggi, ad esempio le tassonomie esterne sono molto più indefinite e difficile da gestire e manutenere, mentre quelle interne sono predefinite e quindi forniscono un riferimento più solido.

Quasi tutti gli elementi del modello informativo di JAXR sono associabili tra di loro. Le associazioni non hanno una tipologia fissa ma si basano sull'astrazione di concetto, rappresentata nel modello informativo di JAXR come `Concept`. Un `Concept` può essere combinato con altri `Concept` per costruire un alberatura che rappresenti una tassonomia.

Gestione del registro

Le JAXR consentono, ovviamente, la gestione delle informazioni presenti nel registro, come la creazione, l'aggiornamento e l'eliminazione di elementi. Inoltre è supportata l'interrogazione del registro per eseguire le ricerche di servizi e aziende.

Ciascun elemento del registro è individuato univocamente da una UUID conforme al DCE 128 bit. Questa chiave è solitamente creata in modo trasparente dal registro anche se per alcuni registri (come ebXML) è possibile fornire dall'esterno la chiave da utilizzare.

Il ciclo di vita degli oggetti del registro inizia con l'operazione di creazione (l'interfaccia `LifeCycleManager` fornisce una serie di metodi di utilità per creare i differenti oggetti definiti nel modello informativo). Un oggetto creato non è ancora presente all'interno del registro e non può dunque essere oggetto di aggiornamenti o cancellazioni. Per inserire l'oggetto nel registro è necessario che questo passi attraverso un'operazione di salvataggio (*save*).

Una caratteristica interessante di JAXR è la possibilità di deprecare gli oggetti, con un concetto simile al tag `@deprecated` di Javadoc. Gli oggetti deprecati non consentono nuove referenze (come associazioni o classificazioni), ma continuano a funzionare normalmente.

Se da una parte l'interfaccia `LifeCycleManager` consente di gestire tutti gli elementi ad alto e basso livello del registro, può essere necessario concentrarsi più su elementi di business. L'interfaccia `BusinessLifeCycleManager` contiene alcune importanti chiamate ad alto livello definendo una API simile alle Publisher API di UDDI. Con questa interfaccia non vengono introdotte nuove funzionalità ma viene allungata una mano agli sviluppatori UDDI che dovrebbero trovarsi a operare con API più familiari. Un limite di JAXR in questa versione è che le operazioni sul ciclo di vita

degli oggetti non sono applicabili a federazioni di connessioni. Non è ad esempio possibile creare la stessa entità direttamente in due registri differenti.

Interrogazione del registro

A differenza delle operazioni "dispositive" (come la creazione o la modifica di informazioni), le operazioni di ricerca avvengono in modo non privilegiato. Non è necessaria dunque l'autenticazione dell'utente.

L'interrogazione può avvenire in modo puntuale o tramite query. Nel primo caso viene utilizzata l'interfaccia `BusinessQueryManager` che, in modo simile a `BusinessLifeCycleManager`, ha una impostazione più ad alto livello e permette l'interrogazione delle interfacce più funzionali del modello informativo.

Molti metodi dell'interfaccia richiedono argomenti simili; alcuni tra i più importanti sono:

- `findQualifiers` :definiscono le regole di confronto di stringhe, ordinamento e operatori logici sui parametri di ricerca.

- `namePatterns`: è una collection di stringhe che contengono nomi, anche parziali con eventuali wildcard come specificato nelle SQL-92 per la parola chiave `LIKE`.

- `classifications`: è una collection di Classification che identifica in quali classificazioni eseguire la ricerca.

Nel secondo caso è possibile utilizzare una query dichiarativa, tramite l'interfaccia `DeclarativeQueryManager`. Ad oggi l'unico standard supportato è una derivazione di SQL-92, in particolare dell'istruzione `SELECT`, estesa con le specifiche relative alle *stored procedures*.

Le query dichiarative sono supportate anche su connessioni federate ma sono una caratteristica opzionale dei provider JAXR 1.0.

Non solo SOAP/WSDL/UDDI

Sebbene questi siano gli standard consolidati per i web service, non sono i soli. Prima di SOAP esisteva XML-RPC, uno standard più semplice per l'esecuzione di chiamate remote tramite XML su Internet. Un'altra iniziativa simile, promossa da OASIS, è ebXML (Electronic Business XML). L'obiettivo di SOAP/WSDL/UDDI è il medesimo di ebXML, ma con un approccio differente: se le tecnologie viste in questo capitolo hanno un approccio "dal basso" (a partire proprio dalla tecnologia), ebXML ha un approccio "dall'alto", ovvero dai requisiti funzionali. Sebbene i due mondi abbiano sviluppato fino ad ora in modo indipendente, si cominciano a notare elementi di convergenza: ebXML utilizza SOAP (esteso con header specifiche e allegati) come protocollo di comunicazione e il consorzio UDDI ha ceduto le specifiche della versione 3.0 al consorzio OASIS. Nell'interesse del successo della visione dei web service, i due mondi, in un futuro non troppo remoto, potrebbero convergerr in uno standard unico.

Bibliografia

Specifiche SOAP: http:/www.w3c.org/TR/SOAP

Specifiche WSDL: http:/www.w3c.org/TR/wsdl

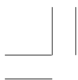
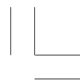
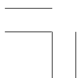
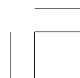
Consorzio UDDI: http:/www.uddi.org

The web Services Tutorial: http://java.sun.com/webservices/tutorial.html

Web Services Made Easier: The Java API & Architectures for XML: http://java.sun.com/xml/webservices.pdf

The Web Services Revolution 1: http://www-106.ibm.com/developerworks/webservices/library/ws-peer1.html

The Web Services Revolution 3: http://www-106.ibm.com/developerworks/webservices/library/ws-peer3/

Capitolo 10

Java Naming and Directory Interface API

GIOVANNI PULITI

JNDI è l'API di Java per i servizi di Naming e Directory e rappresenta lo strumento per poter accedere in modo uniforme e standard ai principali sistemi di naming e directory attualmente a disposizione.

Anche se spesso viene ridotta alla citazione delle due operazioni più ricorrenti, `bind()` e `lookup()`, JNDI è probabilmente l'API più utilizzata all'interno della piattaforma J2EE: tutte le volte che in RMI, in CORBA o in EJB per esempio si effettua una ricerca di un oggetto remoto, in realtà si utilizzano i servizi di base di JNDI. Affrontare nel dettaglio questa API è quindi molto importante non solo per realizzare applicazioni che si interfaccino con sistemi di naming e directory distribuiti, ma anche per comprendere a pieno cosa significhi effettuare un lookup di un oggetto remoto (in RMI) o ricavare la home interface ad esempio di un session enterprise bean (in EJB).

JNDI offre lo stesso livello di astrazione nei confronti dei sistemi di naming che JDBC ha verso i database relazionali. Per questo motivo per poter comprendere a pieno la potenza di questa API, è utile se non indispensabile prima di tutto affrontare i concetti di naming e directory services.

Servizi di naming

Un sistema di naming fornisce il supporto per la gestione di un set di dati associati ad un determinato nome, mentre invece non permette di cercare o manipolare oggetti agendo sui nomi associati. Ogni set di dati deve avere almeno un nome univoco con il quale può essere identificato ad esempio tramite una operazione di *bind* o di *lookup* (più avanti si affronteranno meglio tali concetti).

I servizi di naming sono un pilastro fondamentale di internet, e sono utilizzati ogni giorno da milioni di utenti. Si pensi ad esempio che ogni volta che ci si collega al sito MokaByte, tramite

l'indirizzo del dominio www.mokabyte.it: in realtà si è connessi con il server il cui indirizzo numerico è 213.174.177.203, decisamente più difficile da ricordare. Il computer su cui gira il browser ha effettuato, in modo del tutto trasparente, un'interrogazione su uno dei servizi fondamentali di Internet, il DNS: Domain Naming Service. La domanda era "a quale indirizzo internet è associato il nome www.mokabyte.it?" ed un server predisposto ha fornito la risposta.

Oltre che a rendere la vita più facile per evitare di memorizzare numeri, un servizio di naming porta anche ad una maggior facilità di configurazione di un sistema, permettendo un maggiore disaccoppiamento tra fornitore e fruitore di servizio. Infatti non solo non è necessario sapere che l'indirizzo IP di MokaByte è per l'appunto 213.174.177.203, ma nemmeno essere informati delle operazioni amministrative del service provider di MokaByte, che dall'oggi al domani potrebbe decidere di spostare il suo sito su un indirizzo diverso, per esempio 12.184.31.32. Sarà sufficiente che il database DNS tenga conto della modifica e tutti i lettori di MokaByte potranno continuare a leggere tranquillamente la loro rivista preferita.

Un altro esempio di naming service è quello fornito dal filesystem del sistema operativo: per esempio il nome C:\temp\file.txt (mnemonico e facile da ricordare) è associato ad un ben preciso numero di settore fisico sull'hard disk dove sono memorizzati i dati. Il sistema operativo è libero di riorganizzare il modo in cui memorizza i dati (per esempio una deframmentazione potrebbe spostare i dati su altre aree del disco), ma il nome c:\temp\file.txt permetterà comunque di accedere ai dati senza tenere conto di questi dettagli.

Ogni servizio di naming viene identificato tramite un determinato contesto o context; così il root context è il nome base di una entry, sotto la quale tutte le altre directory sono memorizzate. Ad esempio in un filesystem rappresenta la directory radice (/ in Unix o C:\ in Windows).

Un sottocontesto invece è il nome che rappresenta un set di dati appeso gerarchicamente al contesto principale. Possono esserci più sottocontesti attaccati al contesto principale, in modo da organizzare meglio i vari name-space e conferire al sistema complessivo una maggiore pulizia e flessibilità.

Servizi di directory

Un servizio di directory permette di associare alcune informazioni o metadati, ad un oggetto identificato dal suo nome univoco: è quindi possibile trovare un determinato elemento tramite tali informazioni senza conoscerne il suo nome. In genere un sistema di directory contiene anche un servizio di naming, ma non è detto il viceversa.

Un directory è un particolare tipo di database in grado di organizzare i propri dati in modo gerarchico: fra i più noti si possono sicuramente ricordare LDAP, NIS/NIS+ (Network Information Service di Sun), NDS (Novell Directory Service di Novell), Windows NT Domains, ADS (Active Directory Service).

Con il crescere delle reti informatiche e delle infrastrutture distribuite, è cosa piuttosto frequente che questi sistemi differenti coesistano nello stesso sistema; nasce per questo l'esigenza di accedere con un unico strumento a tali sistemi.

Il protocollo LDAP rappresenta probabilmente il meccanismo migliore per permettere tale unificazione; JNDI è la API Java che permette di parlare LDAP, e per questo è consuetudine

considerare queste due tecnologie strettamente dipendenti, anche se JNDI consente il collegamento con altri sistemi di directory e naming.

LDAP

Il Lightweight Directory Access Protocol (LDAP) è stato sviluppato nei primi anni '90 come standard per la gestione di Directory Services. Tale protocollo non definisce come i dati debbano essere memorizzati sul server, o nel sistema di directory, ma solamente come il client debba accedervi. LDAP fornisce essenzialmente tre tipologie di servizi: Access Control, White Pages Services, Distribuited Computing Directory.

LDAP Access

Si basa essenzialmente su due tipi di servizi, l'autenticazione e l'autorizzazione. L'autenticazione determina l'identità di chi sta utilizzando o desidera utilizzare un determinato software. Anche se non si può essere completamente certi dell'identità di un determinato soggetto, si possono utilizzare differenti meccanismi che permettono a vari livelli di ottenere tale informazione. LDAP fornisce tre livelli di autenticazione: semplice (uid e passwd), per mezzo di certificati digitali (SSL) e tramite il protocollo Simple Authentication and Security Layer (SASL) che rappresenta un misto dei due sistemi precedenti.

Il perché sia necessario utilizzare tali sistemi, e come essi funzionino, è un argomento che esula dagli scopi di questo capitolo anche se dovrebbero in parte essere concetti noti alla maggior parte dei lettori che usano abitualmente sistemi di directory.

Una volta determinata l'identità di un utente, l'autorizzazione definisce quali operazioni siano permesse all'utente e quali sono le risorse alle quali esso può accedere. Si può quindi utilizzare LDAP per definire complesse policies di sicurezza.

LDAP White Pages Services

Questo servizio permette di ricercare persone in un determinato archivio sulla base di precise caratteristiche o attributi, proprio come si può fare ad esempio consultando l'elenco del telefono. Il nome infatti deriva proprio dalle white pages statunitensi, che rappresentano il corrispettivo del nostro elenco del telefono. In genere questo tipo di servizio è reso di pubblico dominio, senza nessuna restrizione di accesso, da tutti i principali sistemi di archivio di schede personali. JNDI in genere partecipa in questo scenario come gateway fra un sistema di anagrafica LDAP ed una applicazione Java vera e propria.

LDAP Distributed Computing Directory

La programmazione distribuita è sicuramente uno dei settori dell'informatica in più ampia e rapida espansione. RMI, CORBA, EJB sono solo alcuni esempi in cui Java è direttamente parte in causa.

In tali scenari il problema principale è determinare dove si trovi un determinato componente di codice e come poterlo ricavare. LDAP da questo punto di vista offre un importante strato di separazione fra una classe ed il suo nome logico e quindi fra il codice remoto e chi lo deve utilizzare. Chiunque abbia realizzato una qualsiasi applicazione RMI o CORBA comprenderà perfettamente cosa significhi tutto ciò e quali siano gli aspetti legati a tali concetti.

In genere tramite LDAP è possibile non solo memorizzare pezzi di codice in modo strutturato in un sistema di directory, ma anche aggiungere alcune informazioni descrittive relative al codice memorizzato, dando luogo a notevoli possibilità aggiuntive.

I dati in LDAP

In LDAP i dati sono organizzati in modo gerarchico secondo uno schema ad albero, detto *Directory Information Tree* (DIT), in cui ogni foglia viene detta entry. La prima entry è la *root entry*. Ogni entry è univocamente identificata da un *Distinguished Name* (DN), più una serie di coppie attributo/valore.

Il DN, che rappresenta l'equivalente della chiave primaria in un database relazionale, indica anche la posizione della entry all'interno dell'albero DIT, come ad esempio il path completo di un file ne identifica la posizione all'interno del file system.
Un esempio di DN potrebbe essere

```
uid=giovanni.puliti, ou=writers, o=mokabyte.it
```

la parte più a sinistra rappresenta il *Relative Distinguished Name* (RDN), ed in questo caso è data dalla coppia uid=giovanni.puliti

Gli attributi LDAP in genere utilizzano valori mnemonici per i vari nomi; ad esempio alcuni tipici attributi potrebbero essere quelli riportati nella tab. 10.1.

Ogni attributo può avere uno o più valori ed più in generale seguire determinate regole secondo quanto specificato in quello che viene detto LDAP *schema*. Lo schema definisce l'objectclasses e gli attributi in un DIT LDAP. Ad esempio un utente può avere uno o più indirizzi di posta elettronica.

L'attributo objectclass che equivale alla tabella di un database relazionale, specifica quali sono gli attributi di una entry, suddividendoli in obbligatori (required) ed opzionali (allowed).

In seguito si ritornerà su questo punto; analizzando l'objectclass `inetOrgPerson`, uno degli oggetti più utilizzati in tutti i sistemi LDAP, potrebbe essere definito nel seguente modo

```
(2.16.840.1.113730.3.2.2
  NAME 'inetOrgPerson'
  SUP organizationalPerson
  STRUCTURAL
  MAY (audio $ businessCategory $ carLicense $ departmentNumber $
    displayName $ employeeNumber $ employeeType $ givenName $
    homePhone $ homePostalAddress $ initials $ jpegPhoto $
```

Tabella 10.1 – *Alcuni esempi di attributi LDAP.*

Attributo LDAP	Definizione	Esempio
cn	common name o nome completo	cn=Giovanni Puliti
sn	surname o soprannome	sn=gsl
givenname	nome	givenname=Giovanni
uid	user id	uid=giovanni.puliti
dn	Distinguished Name	uid=giovanni.puliti, ou=writers, cn=Giovanni Puliti, o=mokabyte, dc=mokabyte, dc=it
mail	email address	mail=gpuliti@mokabyte.it

```
        labeledURI $ mail $ manager $ mobile $ o $ pager $
        photo $ roomNumber $ secretary $ uid $ userCertificate $
        x500uniqueIdentifier $ preferredLanguage $
        userSMIMECertificate $ userPKCS12
    )
)
```

Si noti la prima stringa che rappresenta l'*Object Identifier* o OID, che identifica l'oggetto in modo analogo a come in Java il serialization UID identifica una classe serializzabile.

Il SUP invece identifica il padre di tale oggetto, dato che l'organizzazione gerarchica di LDAP permette di dar vita a gerarchie di oggetti un po' come avviene nel modello OO. In questo caso il processo di generalizzazione-specificazione (legame padre-figlio) permette di aggiungere-rimuovere proprietà e non comportamenti (metodi). Successivamente sono riportati gli attributi obbligatori (campo MUST) e quelli opzionali (MAY).

Il carattere di separazione $ è stato scelto per tradizione, in onore alla convenzione adottata dal protocollo X.500, che, quando fu definito, si basava su uno hardware particolare non in grado di stampare il carattere $, che quindi pur rappresentando il separatore non veniva rappresentato.

Una entry può essere rappresentata anche tramite il formato di interscambio LDIF (LDAP Data Interchange Format), che di fatto è il modo comune con il quale una entry può essere rappresentata in modo leggibile all'occhio umano.

```
version: 1
   dn: cn=Barbara Jensen,ou=Product Development,dc=siroe,dc=com
   objectClass: top
   objectClass: person
   objectClass: organizationalPerson
```

```
objectClass: inetOrgPerson
cn: Giovanni Puliti
displayName: Giovanni Puliti sn: Jensen
givenName: Giovanni
nitials: GP
title: manager, product development
uid: gpuliti
mail: gpuliti@mokabyte.it
telephoneNumber: +39 55 123456789
facsimileTelephoneNumber: +39 55 123456788
mobile: +39 355 123456789
roomNumber: 0209
carLicense: 6ABC246
o: mokabyte
ou: Product Development
departmentNumber: 2604
employeeNumber: 42
employeeType: full time
preferredLanguage: it, en-us;q=0.8, en;q=0.7
labeledURI: http://www.mokabyte.it/redazione/gpuliti
```

Dato che lo scopo di questo capitolo non è una trattazione approfondita di LDAP, si conclude qui questa introduzione a tale protocollo. Per chi fosse interessato a maggiori approfondimenti, si rimanda alla lettura della molta bibliografia presente sull'argomento, fra cui [LDAP].

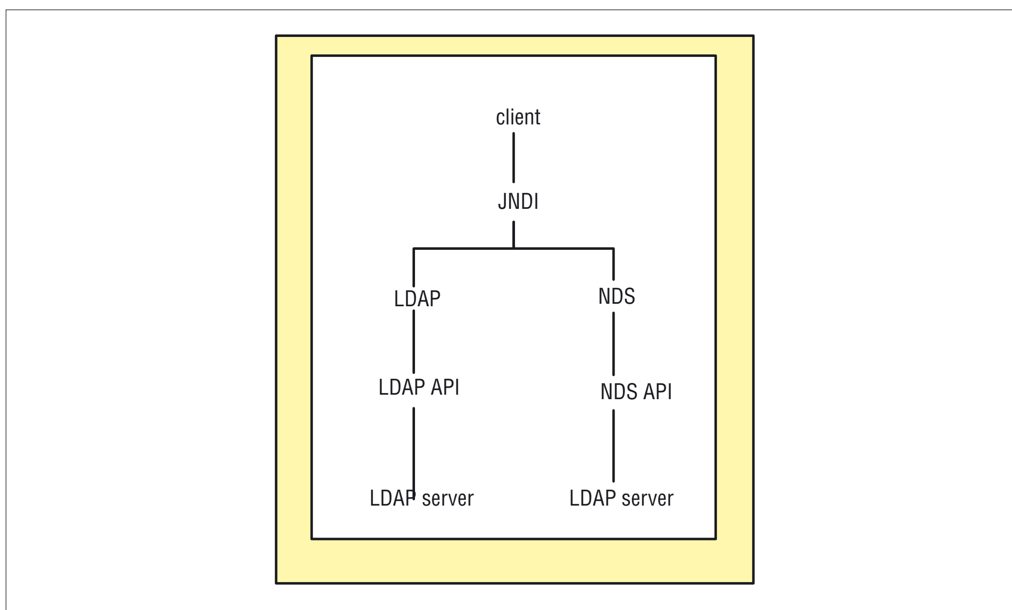
JNDI

Sebbene LDAP stia crescendo in popolarità e diffusione, esso è ancora molto lontano da essere il protocollo universale per tutti i sistemi di naming e directory. Tecnologie come NDS e NIS sono forse più diffusi nel mondo dei server, mentre CORBA, basato su un sistema di naming proprio, era fino a qualche tempo fa la tecnologia più diffusa per la realizzazione di applicazioni distribuite language e platform indipendent; EJB di recente sta velocemente diffondendosi come la tecnologia di riferimento in questo genere di architetture. Per questo JNDI sta diventando sempre più una delle colonne portanti di J2EE ed in particolare di EJB, dove è importantissimo poter disporre di un sistema di localizzazione di oggetti remoti.

Come rappresentato dalla figura 1, si può pensare di utilizzare JNDI per collegarsi non solo ad un server LDAP, ma indifferentemente ad un directory service NDS o NIS. In questo modo JNDI fornisce una interfaccia comune verso i diversi sistemi sottostanti, e si può passare molto facilmente da un sistema all'altro semplicemente cambiando driver.

Purtroppo, differentemente da JDBC, lo strato driver non espone una interfaccia uniforme e completamente standard per cui non sempre è possibile per l'applicazione client astrarsi dai dettagli implementativi e tecnologici sottostanti.

Figura 10.1 – *JNDI può essere utilizzato per interfacciarsi con sistemi di naming/directory differenti, fornendo in questo modo una piattaforma unica.*



Il JNDI Service Provider (JNDI Driver)

Un service provider è un driver che permette di comunicare con un directory service in modo analogo a come un driver JDBC consente la comunicazione con un database relazionale. Come si avrà modo di vedere negli esempi, un driver deve implementare l'interfaccia `Context` o più spesso la `DirContext` che la estende direttamente per accedere ai sistemi di directory.

Il driver però non è in grado di astrarre completamente il sistema sottostante, ma è necessita di alcuni parametri operativi legati al sistema sottostante: ad esempio JNDI non dispone ancora di un linguaggio di interrogazione come invece JDBC che dispone di SQL.

In tal senso il gruppo di sviluppo di XML sta pensando di dar vita a quello che dovrebbe prendere il nome di DSML (Directory Service Markup Language); il lavoro da fare è ancora molto (attualmente è in fase di definizione la specifica DSML 2.0 che dovrebbe aggiungere il supporto per le query e le modifiche sulle directory), e non da tutti ritenuto così importante: si ritiene infatti che gli sforzi maggiori dovrebbero essere volti ad ampliare il supporto per LDAP dei vari sistemi di directory.

Per migliorare l'integrazione di sistemi differenti, JNDI supporta il concetto di federazione in modo che un service provider possa passare una determinata richiesta ad un altro nel caso in cui questa non possa essere soddisfatta dal primo. In questo caso questo meccanismo è

molto simile a quello supportato da JDBC in cui per una connessione al database viene utilizzato il primo driver in grado di comprendere la stringa di connessione.

Il package JNDI viene fornito di default con alcuni service provider per connettersi ai principali sistemi di directory attualmente disponibili: LDAP, NIS, COS (CORBA Object Service), RMI Regostry, File System. Altri provider possono essere utilizzati in alternativa o in aggiunta a quelli di base: ad esempio Novell fornisce un service provider per il collegamento con NDS, mentre IBM e Netscape hanno prodotto alcuni provider alternativi per la connessione con LDAP.

Per chi fosse interessato a sviluppare un proprio service provider può fare riferimento al tutorial JNDI di Sun (vedi [Tutorial]).

Lavorare con JNDI

Le operazioni concesse in JNDI sono essenzialmente le seguenti

- Connessione ad un server LDAP
- Autenticazione sul server (LDAP bind)
- Operazioni sui dati: ricerca, aggiunta modifica e rimozione di una entry
- Disconnessione dal server LDAP

Si vedranno adesso tali operazioni una dopo l'altra

Connessione ed autenticazione

La prima cosa da fare per poter manipolare le informazioni contenute in un archivio LDAP è connettersi con il sistema. Per effettuare tale operazione è necessario ricavare un oggetto che implementa l'interfaccia DirContext. Nella maggior parte dei casi questo può essere fatto tramite l'oggetto InitialDirContext fornendo al costruttore i parametri necessari tramite una hastable. A livello minimale tali parametri comprendono il nome della classe da utilizzare come inizializzatore di contesto, e l'URL del server. Ad esempio si potrebbe scrivere

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://localhost:389");
Context ctx = new InitialContext(env);
```

per la connessione verso un server LDAP in esecuzione sul server locale.

Il parametro Context.INITIAL_CONTEXT_FACTORY è molto importante dato che specifica il tipo di directory service verso il quale si desidera collegarsi. Ad esempio se si volesse utilizzare il file system, si potrebbe scrivere

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.fscontext.RefFSContextFactory");
Context ctx = new InitialContext(env);
```

Utilizzando in questo caso il factory fornito da Sun per la connessione con il directory service filesystem. Da notare che spesso tali parametri possono essere scritti in un file jndi.properties memorizzato nel classpath della applicazione: tale file verrà caricato automaticamente e passato come oggetto Properties al costruttore InitialContext. Normalmente questa operazione viene effettuata nelle applicazioni client di EJB, fornendo i parametri del server e del context Factory.

In questo caso JNDI viene utilizzato per connettersi con un repository di oggetti remoti, non tanto per accedere alle informazioni di un DIT, ma piuttosto per poter ricavare gli stub remoti.

Ecco di seguito alcuni casi per la connessione ai più famosi EJB container disponibili sul mercato:

```
# Weblogic Server
INITIAL_CONTEXT_FACTORY=weblogic.jndi.WLInitialContextFactory
PROVIDER_URL=t3://localhost:7001

# Borland Enterprise Server
INITIAL_CONTEXT_FACTORY=com.inprise.j2ee.jndi.CtxFactory
PROVIDER_URL=iiop:///

# JBoss
INITIAL_CONTEXT_FACTORY=org.jnp.interfaces.NamingContextFactory
PROVIDER_URL=jnp://localhost:1099

# IBM WebSphere
INITIAL_CONTEXT_FACTORY=com.ibm.websphere.naming.WsnInitialContextFactory
PROVIDER_URL=iiop://localhost:900
```

Chi desiderasse approfondire l'utilizzo di JNDI per questo genere di operazioni, può consultare direttamente la specifica di EJB.

Anche se d'ora in poi si focalizzerà l'attenzione su JNDI e LDAP si tenga presente che specificando la classe definita dal parametro INITIAL_CONTEXT_FACTORY è possibile cambiare completamente il tipo di service directory utilizzato. Questa è una operazione molto più potente di quella permessa in JDBC al momento di specificare il driver: tanto per avere un'idea è come se si decidesse di cambiare non solo database server, ma anche passare indifferentemente da un database relazionale ad uno ad oggetti o ad uno di tipo flat file (esempio File .csv).

Al momento della connessione verso il server viene effettuata anche l'autenticazione del client. Questa operazione è detta binding e non deve essere confusa con l'operazione di bind di oggetti con nomi logici come avviene abitualmente con i registry di oggetti.

Nel caso in cui non sia fornita nessuna credenziale, l'utente viene connesso con il profilo anonimo, con un ridotto set di operazioni permesse: ad esempio spesso viene fornito il solo

accesso in lettura verso anagrafiche di utenti. Tutte le operazioni, comprese anche quelle a livello base di lettura, sono definite tramite una ACL definita nel server LDAP.

Ad ogni entry può essere associata una o più ACL in modo da realizzare differenti viste sui dati memorizzati nel DIT. I tre livelli di autenticazione forniti dallo standard LDAP (simple, SSL, SASL), possono essere specificati al momento della connessione utilizzando i relativi parametri al costruttore del contesto. In particolare

- Context.SECURITY_AUTHENTICATION ("java.naming.security.authentication"): specifica il mccanismo di autenticazione da utilizzare. Per il service provider LDAP di Sun, può essere uno dei seguanti valori: *none*, *simple*, *sasl_mech*, dove sasl_mech è una lista separata da spazi di nomi basati sul meccanismo SASL.

- Context.SECURITY_PRINCIPAL ("java.naming.security.principal"): specifica il nome dell'utente o programma client che effettua l'autenticazione. Questo valore dipende dal valore precedente.

- Context.SECURITY_CREDENTIALS ("java.naming.security.credentials"): specifica le credenziali di autenticazione. A seconda del tipo di autenticazione, si può inserire la password o un certificato digitale.

Ecco alcuni esempi di autenticazione

Autenticazione anonima

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://localhost:389/o=JNDITutorial");

// Usa la anonymous authentication
env.put(Context.SECURITY_AUTHENTICATION, "none");

// Crea l' initial context
DirContext ctx = new InitialDirContext(env);

// ... fai qualcosa con ctx
```

Autenticazione simple

```
// Set up the environment for creating the initial context
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://localhost:389/o=JNDITutorial");

// Autentica come utente Giovanni Puliti e password pippo
env.put(Context.SECURITY_AUTHENTICATION, "simple");
```

```
env.put(Context.SECURITY_PRINCIPAL, "cn=Giovanni Puliti, ou=writers,  o=mokabyte");
env.put(Context.SECURITY_CREDENTIALS, "pippo");

// Crea l'initial context
DirContext ctx = new InitialDirContext(env);

// ...fa qualcosa con ctx
```

È possibile eventualmente eseguire una seconda operazione di bind dopo la prima, con un differente livello di autenticazione utilizzando i metodi Context.addToEnvironment() e Context.removeFromEnvironment(). Le operazioni successive lavoreranno con le nuove credenziali fornite.

```
env.put(Context.SECURITY_AUTHENTICATION, "simple");
env.put(Context.SECURITY_PRINCIPAL, "cn=Giovanni Puliti, ou=writers, o=mokabyte");
env.put(Context.SECURITY_CREDENTIALS, "pippo");

// Crea l'initial context
DirContext ctx = new InitialDirContext(env);

// ... fa qualcosa con ctx

// Cambia l' autenticazione usando il profilo anonimo
ctx.addToEnvironment(Context.SECURITY_AUTHENTICATION, "none");

// ... fa qualcosa con ctx
```

In caso di errore durante la procedura di autenticazione si riceve un errore del tipo

```
javax.naming.AuthenticationException: [LDAP: Invalid Credentials]
    at java.lang.Throwable.<init>(Compiled Code)
    at java.lang.Exception.<init>(Compiled Code)
```

Se invece si specifica un tipo di autenticazione non supportata verrà generata una eccezione del tipo AuthenticationNotSupportedException; ad esempio il pezzo di codice seguente

```
env.put(Context.SECURITY_AUTHENTICATION, "myauthentication");
env.put(Context.SECURITY_PRINCIPAL, "cn=Giovanni Puliti, ou=writers,  o=mokabyte");
env.put(Context.SECURITY_CREDENTIALS, "pippo");
```

produce il seguente output:

```
javax.naming.AuthenticationNotSupportedException:
```

```
Unsupported value for java.naming.security.authentication property.
      at java.lang.Throwable.<init>(Compiled Code)
      at java.lang.Exception.<init>(Compiled Code)
      at javax.naming.NamingException.<init>(Compiled Code)
   ...
```

Autenticazione SASL e SSL

Per quanto riguarda i procedimenti di autenticazione tramite SASL e SSL una esauriente e comprensiva analisi di tali meccanismi richiederebbe molto più spazio di quello qui a disposizione e per certi versi esula dagli scopi di questo articolo, per cui si rimanda al tutorial Sun su JNDI [Tutorial], o alla documentazione relativa alla JAAS API.

Leggere le proprietà di una entry

Un primo approccio relativamente alla lettura di informazioni in un DIT può essere fatto tramite il metodo DirContext.getAttributes() che permette di leggere gli attributi di un oggetto nel directory. Il metodo riceve il nome dell'oggetto del quale si vogliono leggere gli attributi, ad esempio per l'oggetto "cn=Giovanni Puliti, ou=writers", si potrebbe scrivere

```
Attributes answer = ctx.getAttributes("cn=Giovanni Puliti, ou=writers ");
```

A questo punto si può ricavare il contenuto della risposta come di seguito

```
for (NamingEnumeration ae = answer.getAll(); ae.hasMore();) {
   Attribute attr = (Attribute)ae.next();
   System.out.println("Stampa dei valori dell' attributo con id: " + attr.getID());
   // Stampa il valore di ogni attributo
   for (NamingEnumeration e = attr.getAll(); e.hasMore();
      System.out.println("- valore: " + e.next()));
}
```

Ricerca di una entry

Dato che l'uso più frequente che si fa di un sistema LDAP è quello di memorizzare informazioni, prima o poi si rende necessario eseguire una ricerca. Si può cercare una entry sulla base del DN, o in base ad ogni tipo di attributo. Le ricerche si effettuano utilizzando una connessione, un punto base da cui iniziare a cercare, definendo la profondità cui effettuare le ricerche ricorsive nel DIT (scope search) e tramite filtri che specificano il criterio di ricerca in modo analogo ai filtri SQL (clausola WHERE). In genere ogni ricerca si basa sulla comparazione di un attributo con un determinato valore: un filtro può essere booleano o di tipo più sofisticato come il "sounds like" implementato da alcuni server.

Tabella10. 2 – *Alcuni esempi di ricerche e filtri da utilizzare.*

Ricerca da effettuare	Filtro da utilizzare
Cerca tutti gli utenti con last name = puliti	sn = puliti
Cerca tutti gli utenti con last name che inizia per pu	sn = pu*
Cerca tutte le entry che hanno la parola administrator nel common name e sono anche del tipo groupOfUniqueNames	(&(cn = *administrator*)(objectclass = grou pofUniqesNames))

Il punto di partenza per la ricerca e la ricorsività nel DIT possono essere specificate tramite appositi attributi: ad esempio se nel DIT rappresentante lo staff di MokaByte si volesse cercare tutti i componenti dello staff si potrebbe specificare come punto di partenza la radice del DIT stesso,

```
dc=mokabyte, dc=it
```

oppure se si volesse restringere la ricerca ai soli autori ed articolisti si potrebbe specificare

```
ou=writers, dc=mokabyte , dc=it
```

In LDAP lo scope di una ricerca indica se tale ricerca deve essere effettuata in modo ricorsivo e la profondità di tale ricorsione. Lo scope può essere specificato tramite i tre attributi raffigurati in Figura 10.2.

In JNDI una ricerca su un DIT LDAP può essere fatta tramite il metodo DirContext.search().

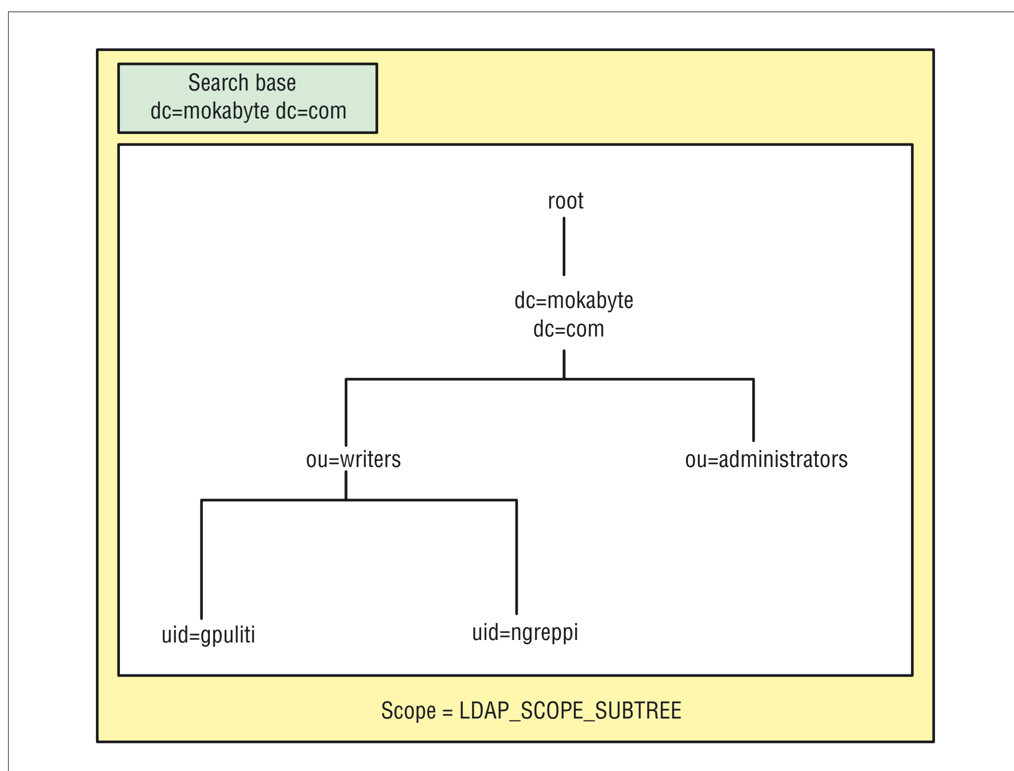
```
// Specifica gli attributi da confrontare
// Cerca gli oggetti con l'attributo "sn" = "gsl"
// e l'attributo "mail" con un valore qualsiasi

Attributes matchAttrs = new BasicAttributes(true);
matchAttrs.put(new BasicAttribute("sn", "gsl"));
matchAttrs.put(new BasicAttribute("mail"));

// Search for objects that have those matching attributes
NamingEnumeration answer = ctx.search("ou=writers", matchAttrs);
```

L'esempio precedente restituisce tutti gli attributi associati alle entries che soddisfano la ricerca. Si può eventualmente selezionare quali attributi debbano essere restituiti come risultato della ricerca passando al metodo search() un array di attributi.

Figura 10.2 – *Alcuni esempi di ricerca in base a punto di partenza e scope.*




```
// Specifica quali attributi debbano essere restituiti
String[] attrIDs = {"sn", "telephonenumber", "mail"};

// Esegue la ricerca
NamingEnumeration answer = ctx.search("ou=People", matchAttrs, attrIDs);
```

Per affinare una ricerca in alternativa all'uso degli attributi, si può utilizzare un filtro, ovvero una espressione logica di confronto. Ad esempio per cercare tutte le entries con un determinato indirizzo di posta elettronica si potrebbe utilizzare la seguente espressione:

```
(&(ou=writers)(mail=*))
```

Il codice seguente crea un filtro ed un default search controls, `SearchControls` e li usa per la ricerca

Tabella 10.3 – *Simboli logici per le comparazioni nelle ricerche*

Simbolo	Descrizione
&	*and* logico
|	or logico
!	Negazione logica
=	Uguaglianza basata sulle regole di confronto dell'attributo
~=	Somiglianza basata sulle regole di confronto dell'attributo
>=	Maggiore di basato sulle regole di confronto dell'attributo
<=	Minore di basata sulle regole di confronto dell'attributo
=*	Presenza di un attributo indipendentemente dal valore
*	wildcard (zero o più caratteri)
\	escape (per i caratteri escaping '*', '(', o ')'

Tabella 10.4 – *Attributi dello scope di una ricerca*

Scope	Significato
LDAP_SCOPE_SUBTREE	Inizia la ricerca dal punto base e prosegue in profonfità scendendo fino all'ultimo livello.
LDAP_SCOPE_ONLEVEL	Inizia la ricerca dal punto base e prosegue in profonfità scendendo di un livello.
LDAP_SCOPE_BASE	Ricerca solo al livello del punto base

```
// Crea  i default search controls
SearchControls ctls = new SearchControls();

// Specifica il filtro di ricerca
String filter = "(&(ou=writers)(mail=*))";

// Cerca utilizzando il filtro
NamingEnumeration answer = ctx.search("o=mokabyte", filter, ctls);
```

La sintassi e il significato delle espressioni logiche utilizzabili dovrebbe essere piuttosto intuitiva. Nella tab. 10.3 è riportata una breve spiegazione. Per una completa descrizione si faccia riferimento alla RFC 2254.

Lo scope di una ricerca indica la profondità con cui una ricerca deve essere effettuata ricorsivamente nell'albero del DIT. In LDAP si possono avere i valori riportati nella tab. 10.4.

Ecco un esempio di utilizzo degli attributi di scope nelle ricerche:

```
String[] attrIDs = {“sn”, “telephonenumber”, “mail”};
SearchControls ctls = new SearchControls();
ctls.setReturningAttributes(attrIDs);
ctls.setSearchScope(SearchControls.SUBTREE_SCOPE);
// Specifica il filtro di ricerca
// cerca gli oggetti che hanno attributo “sn” == “gsl”
// ed un valore qualsiasi in “mail”
String filter = “(&(sn=gsl)(mail=*))”;
NamingEnumeration answer = ctx.search(“”, filter, ctls);
```

Una ricerca può essere limitata nel numero massimo di elementi ritornati, o nel tempo massimo di esecuzione. Nel primo caso si può utilizzare il metodo SearchControls.setCountLimit(), come ad esempio

```
SearchControls ctls = new SearchControls();
ctls.setCountLimit(1);
```

Se il programma cerca di ottenere più risultati del numero massimo specificato, verrà generata una SizeLimitExceededException. Il limite temporale invece può essere specificato tramite il metodo SearchControls.setTimeLimit(); anche in questo al superamento del limite caso verrà generata una eccezione TimeLimitExceededException.

```
SearchControls ctls = new SearchControls();
ctls.setTimeLimit(1000);
```

Impostare il limite temporale a zero equivale a non imporre nessuna restrizione.

Modificare gli attributi di una entry

L’interfaccia DirContext offre alcuni metodi per la modifica degli attributi e dei loro valori. Un modo per effettuare tali modifiche è fornire una lista di ModificationItem che rappresentano le costanti indicanti il tipo di modifica che si desidera fare. I valori consentiti sono i seguenti:

```
ADD_ATTRIBUTE
REPLACE_ATTRIBUTE
REMOVE_ATTRIBUTE
```

Il cui significato dovrebbe essere piuttosto intuitivo. Le modifiche sono ovviamente applicate nell’ordine in cui compaiono nella lista e sono eseguite tutte oppure nessuna.

Il seguente codice di esempio esegue una serie di modifiche su una entry: per prima cosa modifica l’indirizzo di posta elettronica (attributo mail), aggiunge un nuovo numero di telefono (attributo telephonenumber) e rimuove l’attributo codfiscale.

```
// Specifica le modifiche da effettuare
ModificationItem[] mods = new ModificationItem[3];

// Modifica l'attributo "mail"  con un nuovo valore
mods[0] = new ModificationItem(DirContext.REPLACE_ATTRIBUTE,
new BasicAttribute("mail", "gpuliti@mokabyte.it"));

// aggiunge un valore addizionale all'attributo "telephonenumber"
mods[1] = new ModificationItem(DirContext.ADD_ATTRIBUTE,
new BasicAttribute("telephonenumber", "123456789"));

// rimuove l'attributo  "codfiscale"
mods[2] = new ModificationItem(DirContext.REMOVE_ATTRIBUTE, new BasicAttribute("codfiscale"));
```

Non sempre è possibile inserire valori multipli per ogni attributo, dato che non tutti i server supportano questa caratteristica: in questo caso si dovrà quindi rimpiazzare DirContext.ADD_ATTRIBUTE in DirContext.REPLACE_ATTRIBUTE per il telephonenumber.
Si consulti a tal proposito la documentazione del prodotto utilizzato. Dopo aver creato la lista delle modifiche le si potranno eseguire tramite

```
ctx.modifyAttributes(name, mods);
```

Memorizzare oggetti nel directory

Nel caso in cui si lavori con applicazioni Java distribuite, uno dei problemi da risolvere è consentire ai vari client di utilizzare oggetti remoti in modo condiviso. La soluzione più semplice per risolvere questo aspetto è utilizzare un directory service, memorizzando tali oggetti al suo interno, in modo che tutti possano localizzarli e ricavarli.

Esistono sostanzialmente tre modi per inserire un oggetto in un directory service: memorizzare una copia dell'oggetto stesso, un suo reference, o l'insieme degli attributi che lo descrivono. Ad esempio utilizzando la serializzazione Java, si può memorizzare sia lo stato dell'oggetto, mentre il suo reference non è altro che una rappresentazione compatta dell'indirizzo di memoria utilizzabile per ricavare l'oggetto. Infine rappresentare l'oggetto tramite i suoi attributi significa utilizzare una collezione di proprietà che ne descrivono lo stato, compreso il suo indirizzo e le informazioni sullo stato.

Scegliere quale forma di rappresentazione utilizzare dipende principalmente dal tipo di oggetto in esame e dal sistema sottostante, ma anche dal tipo di applicazione che si deve realizzare e dalla particolare modalità con cui si desidera interagire con gli oggetti.

I metodi utilizzati per memorizzare un oggetto sono i seguenti

```
Context.bind()(in the API reference documentation)
DirContext.bind()(in the API reference documentation)
```

```
Context.rebind()(in the API reference documentation)
DirContext.rebind()(in the API reference documentation)
```

L'oggetto passato a questi metodi verrà trasformato in una forma equivalente all'interno del directory, in funzione del tipo di oggetto e del supporto offerto dal directory per quell'oggetto.

Si vedrà ora come memorizzare nel directory oggetti serializzabili, reference di oggetti, oggetti con attributi.

Mentre si tralasceranno i dettagli relativi alla gestione degli oggetti remoti RMI (inclusi quelli che utilizzano IIOP) e gli oggetti CORBA, per i quali sicuramente è necessaria una trattazione più ampia delle tecnologie relative. Per chi fosse interessato a maggiori approfondimenti relativi a queste tecnologie e soprattutto alla memorizzazione degli oggetti nel directory, si rimanda alla letteratura ufficiale, nonché ai capitoli del libro che li trattano (RMI e CORBA).

Oggetti serializzabili

Il processo di serializzazione Java permette di trasformare un oggetto in uno stream di byte in modo che il processo inverso possa essere effettuato partendo direttamente da tale stream. Il runtime Java fornisce il supporto per il processo di serializzazione e deserializzazione standard, anche se è possibile personalizzare tali trasformazioni, variando quindi il modo con cui i dati sono estrapolati da un oggetto ed immessi nello stream.

Importante ricordare che durante la serializzazione di un oggetto esso viene trasformato in una sequenza di byte, prelevando dal suo interno i suoi dati ed il nome della classe. L'oggetto in se non verrà memorizzato nello stream, ma solamente il suo codice univoco, il serialVersionUID. Tramite tale codice al momento della deserializzazione, la JVM potrà ricavare il nome esatto della classe da istanziare, che dovrà essere fornita alla JVM tramite classpath. L'operazione di bind (registrazione nel directory) di un oggetto serializzabile avviene tramite i metodi DirContext.bind() nel caso di un nuovo bind, oppure Context.rebind() e DirContext.rebind() per le operazioni di riscrittura.

Se ad esempio si disponesse di un oggetto serializzabile MySerial, si potrebbe scrivere

```
String ObjectName = "Oggetto serializzabile numero 1";
MySerial ms = new MySerial (ObjectName);
// Esegue la bind
ctx.bind("cn=MySerial", ms);
```

L'operazione di ricerca a questo punto può avvenire banalmente nel seguente modo

```
// Ricava l'oggetto dal directory
MySerial ms2 = (MySerial)ctx.lookup("cn=MySerial");
…
// Fai qualcosa con ms2
```

Durante le operazioni di bind e lookup l'oggetto deve essere rintracciabile dalla JVM, ovvero deve essere presente nel classpath.

In alternativa si può pensare di utilizzare il suo codebase direttamente al momento della bind: questa tecnica può risultare utile in quei casi in cui l'oggetto risulti non direttamente accedibile nel classpath locale oppure quando il sistema sottostante non supporta l'accesso a tale variabile d'ambiente.

Per memorizzare nel repository anche il codebase dell'oggetto si può procedere direttamente al momento della bind, o successivamente specificando un attributo DirContext.modifyAttributes().

Anche se non obbligatorio, l'etichetta alla quale si associa il codebase è normalmente javaCodebase (attributo specificato in RFC 2713), il quale deve puntare all' URL del codebase della classe, o in forma di directory o di JAR file; se il codebase contiene più URL, questi dovranno essere separati da uno spazio. Lo stesso esempio di prima potrebbe quindi essere riscritto nel seguente modo

```
String ObjectName = "Oggetto serializzabile numero 1";
MySerial ms = new MySerial (ObjectName);
// Esegue la bind utilizzando il codebase dell'oggetto MySerial
ctx.bind("cn= MySerial ", ms, new BasicAttributes("javaCodebase", codebase));
```

In questo caso il codebase dell'oggetto MySerial potrebbe puntare ad un file server accessibile tramite il protocollo HTTP. Differentemente dal caso precedente (bind del nome dell'oggetto), dopo l'operazione di bind, il file MySerial.class non sarà più necessario e potrà essere rimosso dal file system.

Oggetti Referenceable e Reference

In alcuni casi memorizzare oggetti in forma serializzata può non essere la soluzione migliore: la forma serializzata potrebbe essere troppo ingombrante, potrebbe essere riduttiva o non adatta per le informazioni che realmente si vogliono memorizzare, o infine perché il sistema di directory non supporta tale meccanismo. In questi casi si può passare a memorizzare l'oggetto nel directory utilizzando il suo reference. Formalmente un reference, rappresentato dalla classe Reference, è costituito da una lista ordinata di indirizzi di memoria ed informazioni sulla classe che rappresenta l'oggetto in questione. Ogni indirizzo è rappresentato dalla classe RefAddrs e contiene alcune informazioni su come costruire l'oggetto.

In genere un reference è utilizzato per rappresentare oggetti di alto livello in modo da tralasciare i dettagli implementativi, come nel caso delle connessioni di rete, database o file system. Vedendo le cose da un altro punto di vista si potrebbe dire che tali oggetti non sono altro che interfacce o puntatori ad aree di memoria; un puntatore ad una connessione JDBC o un socket, tutte cose che in genere sono gestite direttamente dal sistema sottostante o direttamente dal sistema operativo; in questi casi Java partecipa come sistema di astrazione per ciò che sta sotto. Fra i vari indirizzi di memoria che sono contenuti in un reference ci possono

essere anche puntatori alla classe che rappresenta tale oggetto o ad un particolare factory necessario per ricrearlo.

Un oggetto referenceable implementa l'interfaccia Referenceable la quale tramite il metodo getReference() restituisce il reference all'oggetto. Si potrebbe quindi pensare di scrivere

```
public class MyReferenceableClass implements Referenceable {
  String Name;
  public MyReferenceableClass (String n) {
    Name = n;
  }
  public Reference getReference() throws NamingException {
    return new Reference(MyReferenceableClass.class.getName(), new StringRefAddr("name", Name),
                         MyFactory.class.getName(), somewhere);
  }
  public String toString() {
    return name;
  }
}
```

In questo caso il reference di una istanza di MyReferenceableClass consiste di un indirizzo (rappresentati dalla classe StringRefAddr) che contiene il tipo dell'oggetto, ovvero in questo caso la variabile Name. Da notare l'utilizzo del Factory necessario in seguito per ricreare l'oggetto. Più semplicemente in questo caso, se si creasse

```
MyReferenceableClass mrc = new MyReferenceableClass("myObject");
```

Allora l'address type sarebbe "name", ed il suo valore "myObject". Infine a questo punto si potrebbe scrivere

```
MyReferenceableClass mrc = new MyReferenceableClass("myObject");
// esegue la bind
ctx.bind("cn=mypersonalobject", mrc);
```

l'esecuzione dei metodi bind() e rebind(), o meglio la loro specifica implementazione all'interno del service provider, provocano l'estrazione del reference dell'oggetto (ricavato dal metodo Referenceable.getReference()) e la successiva memorizzazione nel directory.

Al momento della operazione simmetrica di lookup il reference verrà nuovamente convertito nell'oggetto corrispondente tramite l'apposito factory specificato.

```
// ricava l'oggetto indietro
MyReferenceableClass anothermrc = (MyReferenceableClass ) ctx.lookup("cn=mypersonalobject");
...
```

Rispetto all'utilizzo di oggetti serializzabili si può notare come, anche in questo caso, dal punto di vista del client le operazioni di bind e lookup siano del tutto analoghe fra loro. Infine da tenere presente che ovviamente si possono memorizzare oggetti referenceable nel directory solo se il service provider utilizzato supporta entrambi gli oggetti References e Referenceable.

Oggetti con attributi

Se l'oggetto che si vuole memorizzare nel directory non è ne serializzabile e nemmeno referenceable si può procedere ad una forma differente di trasformazione, basata sui suoi attributi, a patto che il sistema sottostante permetta di effettuare bind di oggetti DirContext.

In questo caso la procedura base è fornire all'oggetto gli strumenti per estrapolare in modo autonomo tutte le informazioni descrittive dell'oggetto e di fornirle al servizio di directory tramite un oggetto apposito; questa funzionalità è possibile se l'oggetto implementa l'interfaccia DirContext e se ridefinisce il metodo getAttributes().

Ad esempio un oggetto di questo genere potrebbe essere

```
public class MyClassWithAtributes implements DirContext {

  String Name;
  public MyClassWithAtributes (String n) {
    Name = n;
    myAttrs = new BasicAttributes(true);
    Attribute oc = new BasicAttribute("objectclass");
    oc.add("extensibleObject");
    oc.add("top");
    myAttrs.put(oc);
    myAttrs.put("myobjectName", n);
  }
  public Attributes getAttributes(String name) throws NamingException {
    return (Attributes)myAttrs.clone();
  }

  public String toString() {
    return type;
  }
}
```

Il server al momento della memorizzazione all'interno del directory service, estrapolerà tutte le informazioni che verranno fornite dal metodo getAttributes(). In questo modo di fatto è il programmatore che definisce l'oggetto, che decide cosa e come debba essere memorizzato nel

directory. Tale soluzione risulta essere per questo motivo molto potente e consente di ottimizzare lo spazio utilizzato da ogni oggetto e di conseguenza anche le performance complessive.

Anche in questo caso il client dovrà al solito eseguire semplicemente una bind ed una lookup successivamente.

Object Factory

Fino a questo momento si è potuto vedere in più di una occasione che l'operazione di lookup di un oggetto memorizzato nel directory, implichi internamente la sua ricreazione e la successiva restituzione al client. Tale ricreazione non può avvenire in modo automatico e trasparente ma si rende necessario un apposito factory dell'oggetto in questione.

Affrontare in modo esauriente e completo tutti gli aspetti legati ai factory richiederebbe molto più spazio di quello qui a disposizione, per cui ci si limiterà a dare alcuni esempi limitatamente a quanto visto fino a questo punto.

Un factory, come lascia intuire il nome, è un produttore di oggetti, che accetta in input una serie di informazioni necessarie per poter ricostruire correttamente l'oggetto.

Tutte le informazioni necessarie ed i criteri necessari per validare tali dati sono relative al factory in questione e quindi al tipo di oggetto utilizzato. Ad esempio nel caso degli oggetti referenceable l'esecuzione di una bind da vita ad una serie di operazioni di preparazione necessarie per poter effettuare correttamente la lookup. In particolare in questa fase il service provider utilizzato invoca il metodo DirectoryManager.getObjectInstance() per ricavare il reference individuato dal nome logico passato alla invocazione della lookup. A questo punto a partire dal reference si procede a ricavare il factory specifico per quell'oggetto, e successivamente ad invocarne il metodo getObjectInstance(). Nel caso visto in precedenza, MyReferenceableClass, il factory potrebbe essere qualcosa di molto semplice. Il metodo getObjectInstance prima verifica che i dati passati siano corretti ed utilizzabili per ricreare l'oggetto, ovvero che i dati contenuti nel Reference contengano un address mrc (dove mrc è l'istanza di MyReferenceableClass precedentemente inserita nel directory) e che l'oggetto contenuto sia di tipo MyReferenceableClass. Se tale verifica fallisce, allora il factory restituisce null, in modo che nessun altra operazione di factory possa essere tentata; infatti la ricreazione di un oggetto segue in modo simile il meccanismo a cascata di ricreazione di una connessione JDBC a partire dal nome del driver (il primo driver che corrisponde al tipo di connessione viene utilizzato).

Se invece i controlli avvengono con successo, l'address contenuto (nell'esempio precedente era myObject) viene utilizzato per ricreare l'oggetto.
Il metodo getObjectInstance() di MyReferenceableClassFactory potrebbe essere così definito

```
public Object getObjectInstance(Object obj, Name name, Context ctx, Hashtable env) throws Exception {
  if (obj instanceof Reference) {
    Reference ref = (Reference)obj;
    if (ref.getClassName().equals(Fruit.class.getClassName())) {
      RefAddr addr = ref.get("mrc");
      if (addr != null) {
```

```
                return new MyReferenceableClass ((String)addr.getContent());
              }
           }
        }
     return null;
      }
```

Come si può notare il codice è molto semplice, anche se concettualmente molto importante. Per chi fosse interessato ad ulteriori informazioni o altri esempi di factory differenti potrà fare riferimento al tutorial su JNDI fornito da Sun (vedi [Tutorial]).

Conclusione

Come si è potuto vedere JNDI offre un potente sistema di gestione di archivi gerarchici permettendo tutte le operazioni che similmente sono eseguibili in un archivio gerarchico.

La API JNDI offre molti e potenti meccanismi sia per la gestioni di strutture gerarchiche di nomi che di oggetti.

Si sarà notato come molte delle nozioni affrontate sono spesso legate ad altre API e tecnologie di J2EE, a dimostrazione dell'importanza di questa API. Una volta di più è bene ricordare che JNDI, benché spesso trascurata dai vari testi su J2EE, rappresenta una delle colonne portanti non solo di tutto J2EE ma anche dell'intera piattaforma Java2.

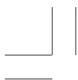
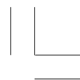
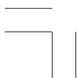
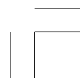
Una sua corretta conoscenza è quindi un fattore indispensabile per ogni programmatore che desideri maneggiare con sicurezza oggetti come RMI, CORBA, EJB, ma anche JSP o JDBC.

Bibliografia

[RFC2798] *Definition of the inetOrgPerson LDAP Object Class,*
http://www.faqs.org/rfcs/rfc2798.html

[Tutorial] *The JNDI Tutorial*,
http://java.sun.com/products/jndi/tutorial

[LDAP] Brad Marshall, *Introduction to LDAP,*
http://quark.humbug.org.au/publications/ldap_tut.html

Capitolo 11

Java Connector Architecture

STEFANO ROSSINI – GIANLUCA MORELLO

Introduzione

Data la presenza congiunta di pacchetti applicativi acquisiti dal mercato, di applicazioni legacy e di applicazioni sviluppate in outsourcing, tutti gli scenari enterprise devono affrontare il problema dell'eterogeneità. Oltre a queste problematiche tradizionali, gli scenari moderni devono affrontare e indirizzare in maniera organica i nuovi requirement portati dall'e-commerce/e-business: comunicazione B2B, apertura verso nuovi canali, estensione di funzionalità, aumento dei livelli di servizio (per esempio: 7 × 24), comunicazioni con info providers, on-line trading broker etc.

È dunque indispensabile per l'azienda costruire un'infrastruttura IT capace di assorbire in maniera flessibile l'impatto delle nuove tecnologie, salvaguardando il patrimonio informativo esistente e rispondendo alle necessità di comunicazione real-time fra le differenti componenti dei sistemi informativi.

La somma di questi fattori porta i principali analisti a considerare l'Enterprise Application Integration (EAI) come uno dei settori fondamentali del mercato software e come uno dei fattori di successo per tutte le aziende che, a partire da una consistente realtà enterprise, intendono adottare nuovi modelli di business e competere sul mercato. Integrare sistemi ed applicazioni sviluppati e progettati da team differenti, in tempi differenti e con l'adozione di tecnologie e paradigmi differenti è chiaramente un compito arduo.

Tipicamente l'integrazione si applica all'idea di Enterprise Information Systems (EIS). Con il termine EIS, si identifica un sistema che fornisce un'infrastruttura di informazioni per un'applicazione Enterprise. Esempi tipici di EIS sono i sistemi legacy, le basi dati (relazionali Oracle, SQLServer, MySQL, …), i sistemi ERP (Enterprise Resource Planning), i sistemi CRM (Customer Relationship Management), e così via.

Il mercato dei tool EAI è molto complesso e offre un set vastissimo di famiglie di prodotti non facilmente comparabili, di difficile utilizzo e soprattutto totalmente dominato da soluzio-

ni proprietarie. La specifica Java Connector Architecture (JCA), seppur orientata al solo ambito J2EE, assume in questo scenario una grande rilevanza in quanto è uno dei primi tentativi di stabilire uno standard nel campo dell'integrazione di EIS.

Integrazione e comunicazione

Integrare applicazioni significa essenzialmente portare sistemi disegnati e sviluppati indipendentemente a lavorare insieme. Lo scambio di informazioni tra sistemi informativi eterogenei e autonomi richiede un livello di comunicazione e varie tipologie di servizi atti a gestire e automatizzare lo scambio di grandi quantità di informazione.

L'incompatibilità di base dei vari sistemi informativi, la presenza di un gran numero di protocolli, di linguaggi, di applicazioni e di piattaforme rende problematica la comunicazione e l'integrazione tra differenti EIS.

Le esigenze di comunicazione e integrazione sono spesso affrontate con meccanismi di comunicazione asincrona mediante processi batch. Questo comporta alti costi e forti rallentamenti nell'attivazione di nuovi canali di comunicazione (per esempio: esposizione Internet di applicazioni legacy) e sostanzialmente limita la definizione e il management di processi di business interaziendali real-time e di alto livello.

Questi limiti rendono ad esempio molto onerosa l'integrazione di Sistemi Informativi a seguito di operazioni di acquisizione/fusione aziendali, portando nella quasi totalità delle situazioni a integrazioni assolutamente verticali di difficile manutenzione/evoluzione. In questi scenari sono di grande aiuto le funzionalità tipiche dei prodotti EAI.

Funzionalità base EAI

All'interno dello scenario di soluzioni EAI, sono individuabili alcune funzionalità/problematiche base:

- *Resource Adapter (RA)*: risolvono i problemi di comunicazione. Quasi tutti i fornitori EAI forniscono Adapter proprietari (sincroni e asincroni) per la comunicazione con specifici EIS.

- *Data Mapping*: tipicamente i dati acquisiti da un RA in un formato proprietario devono essere trasformati secondo le esigenze del business object richiedente. Di solito è l'operazione più costosa nell'integrazione di EIS.

- *Messaging brokers*: tipicamente opera come messaging layer e può operare in modalità point-to-point e publish/subscribe.

- *Workflow*: in questo contesto si intende il coordinamento di business object e messaggi nell'esecuzione di processi di business.

Queste funzionalità EAI vengono tipicamente fornite dagli Integration Middleware. Esempi di questa tipologia di middleware sono gli Integration Broker (IBM MQSeries Integrator, Tibco ActiveEnterprise, Microsoft BizTalk Server, …), i Business Process Manager (IBM MQSeries Workflow, iPlanet ProcessBuilder, Vitria Technology BusinessWare Automator), i Gateways (Database, MOM e Platform) e gli Adapter.

Rispetto a questo scenario, lo scope di JCA a oggi è sostanzialmente limitato al mondo degli adapter e copre quindi una parte relativamente piccola dell'intera piattaforma d'integrazione. La maggior parte delle funzionalità appena discusse e alcuni pattern tipici d'integrazione non sono ancora indirizzati da J2EE. Mancano ad esempio il supporto per il workflow, il data mapping (anche se J2EE fornisce per i DB qualcosa di simile con gli EJB CMP), il supporto per la trasformazione semantica dei dati.

Per affrontare con successo uno scenario di integrazione enterprise, la tecnologia e le funzionalità appena citate non sono sufficienti: è necessario un opportuno approccio metodologico.

Metodologie di integrazione: approccio spaghetti vs. approccio centralizzato

Affrontare problematiche di integrazione complesse all'interno di uno scenario enterprise non è semplicemente un problema tecnologico, ma ha pesanti implicazioni metodologiche ed organizzative.

Nella maggior parte degli scenari attuali, l'integrazione tra programmi e dati è fatta tramite batch file transfer invece che con meccanismi di accesso in tempo reale. Si implementano cioè connessioni punto a punto (via specifiche API) senza utilizzare un'architettura/infrastruttura organica e centralizzata di integrazione. Questo approccio viene definito "spaghetti integration".

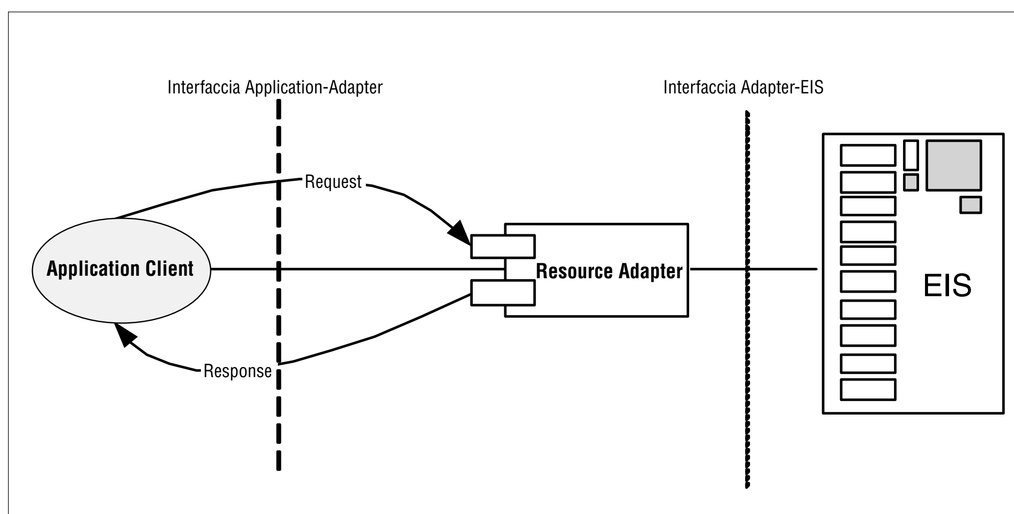
Operare l'integrazione tra applicazioni in un contesto simile comporta complessità e costi di manutenzione ed evoluzione via via crescenti, senza alcuna garanzia di raggiungere i risultati attesi.

L'adozione di tecnologie più sofisticate quali sistemi di messaging, gateway e middleware di comunicazione, consente di semplificare le problematiche di comunicazione e supporta meccanismi sincroni e real-time. In assenza di una struttura centralizzata permanente e di una visione consistente delle problematiche di integrazione, la semplice adozione di queste tecnologie non riesce ad abbattere i costi e le complessità della "spaghetti integration".

L'approccio metodologico corretto presuppone una gestione organica e centralizzata delle problematiche di integrazione. Analizzata in questi termini, l'integrazione non è più un problema del singolo gruppo di sviluppo, ma un problema comune che coinvolge le applicazioni dei vari dipartimenti, data center, uffici e partner. Come tale va affrontata, definendo un'infrastruttura centralizzata presidiata da un team specialistico su cui andranno a interagire tutte le applicazioni.

L'elemento cardine dell'infrastruttura è l'integration broker. Questo componente deve essere l'interfaccia unica attraverso cui transita ogni richiesta di comunicazione tra sistemi applicativi disomogenei. L'integration broker è il punto di normalizzazione tra i vari sistemi applicativi. Le funzionalità principali di un integration broker sono l'intelligent routing e la trasformazione sintattica e semantica dei messaggi.

Figura 11.1 – *Il Resource Adapter.*



L'adapter

Comunque si operi, il primo problema che è necessario affrontare in ambito integrazione è essenzialmente un problema di comunicazione. Sia che si adotti una visione centralizzata, sia che si operi punto-punto il problema principale è la normalizzazione del protocollo specifico della risorsa con un'interfaccia fruibile dal chiamante. Come si è detto, il problema viene tipicamente affrontato attraverso componenti denominati genericamente adapter. Nel caso specifico di adapter destinati alla comunicazione con risorse EIS, vengono più precisamente detti Resource Adapter (RA).
Il RA è un componente che espone delle interfacce di accesso (client API) per permettere la comunicazione con la corrispondente risorsa EIS.

Il RA ha quindi un duplice scopo:

- fornire un'interfaccia omogenea ai client per accedere all'EIS;
- mascherare la complessità dell'interazione con l'EIS al client.

Il RA può essere sincrono o asincrono. Nel caso sincrono, il client, dopo aver invocato un metodo del RA, inoltra la request e si pone in attesa della relativa response (modello RPC). Nel caso asincrono, il client prosegue la sua elaborazione senza attendere l'arrivo della response (modello messaging). Lo standard JCA 1.0 consente esclusivamente l'integrazione mediante comunicazione sincrona. La versione 1.5 dello standard introduce il supporto a RA asincroni.

Che cosa è JCA?

JCA è l'acronimo di Java Connector Architecture. Definisce un'architettura standard [SPEC] per uniformare l'accesso a EIS eterogenei all'interno di applicazioni J2EE. Si focalizza sul design di adapter software destinati a connettere applicazioni J2EE con applicazioni non-Java, risorse legacy e pacchetti applicativi.

L'architettura JCA è costituita da tre elementi: le API Common Client Interface (CCI), i Resource Adapter e i System Contracts. CCI definisce un insieme di API per uniformare l'accesso a EIS da parte di applicazioni J2EE. Grazie a CCI non è necessario affrontare un'integrazione ad hoc per ogni tipologia di EIS, analogamente a quanto avviene con JDBC, JNDI e JMS.

Il connettore vero e proprio (Resource Adapter) viene fornito dai produttori di sistemi EIS. Il Resource Adapter (RA) può essere invocato mediante le API CCI e deve essere installato (modello plugin) all'interno dell'Application Server J2EE.

Un mercato di Adapter standard preconfezionati, *pluggable* nell'infrastruttura J2EE potrebbe consentire la riduzione dei tempi e dei costi di sviluppo dell'integrazione, riducendo il time-to-market (tanto caro ai "commerciali") e indirizzando soluzioni standard scalabili, sicure e transazionali.

Figura 11.2 – *L'architettura JCA.*

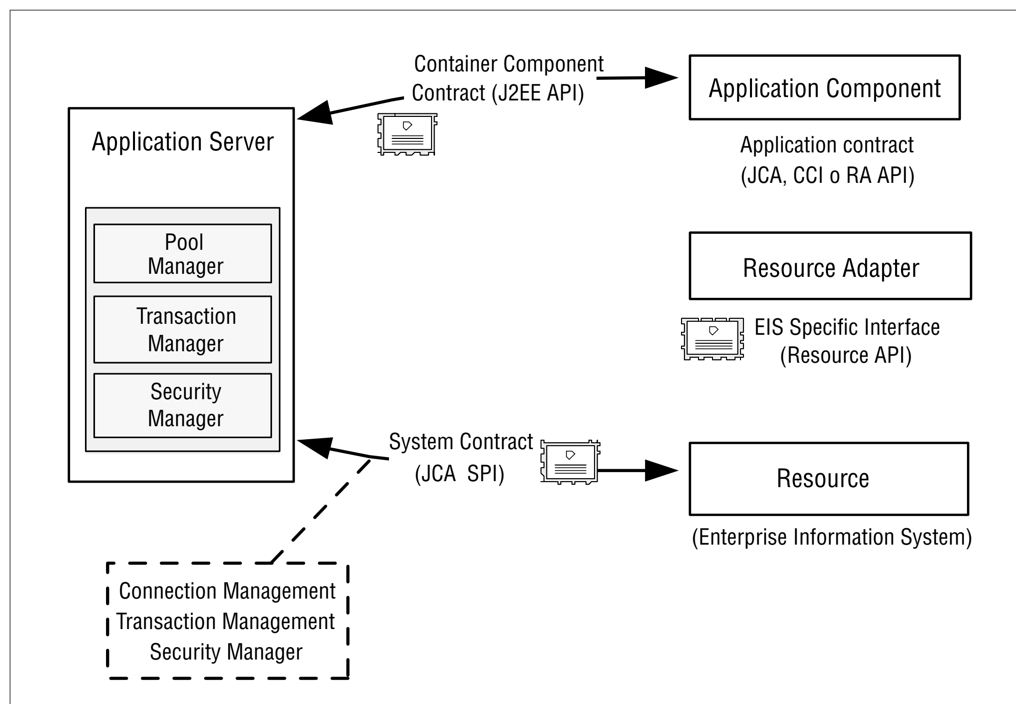
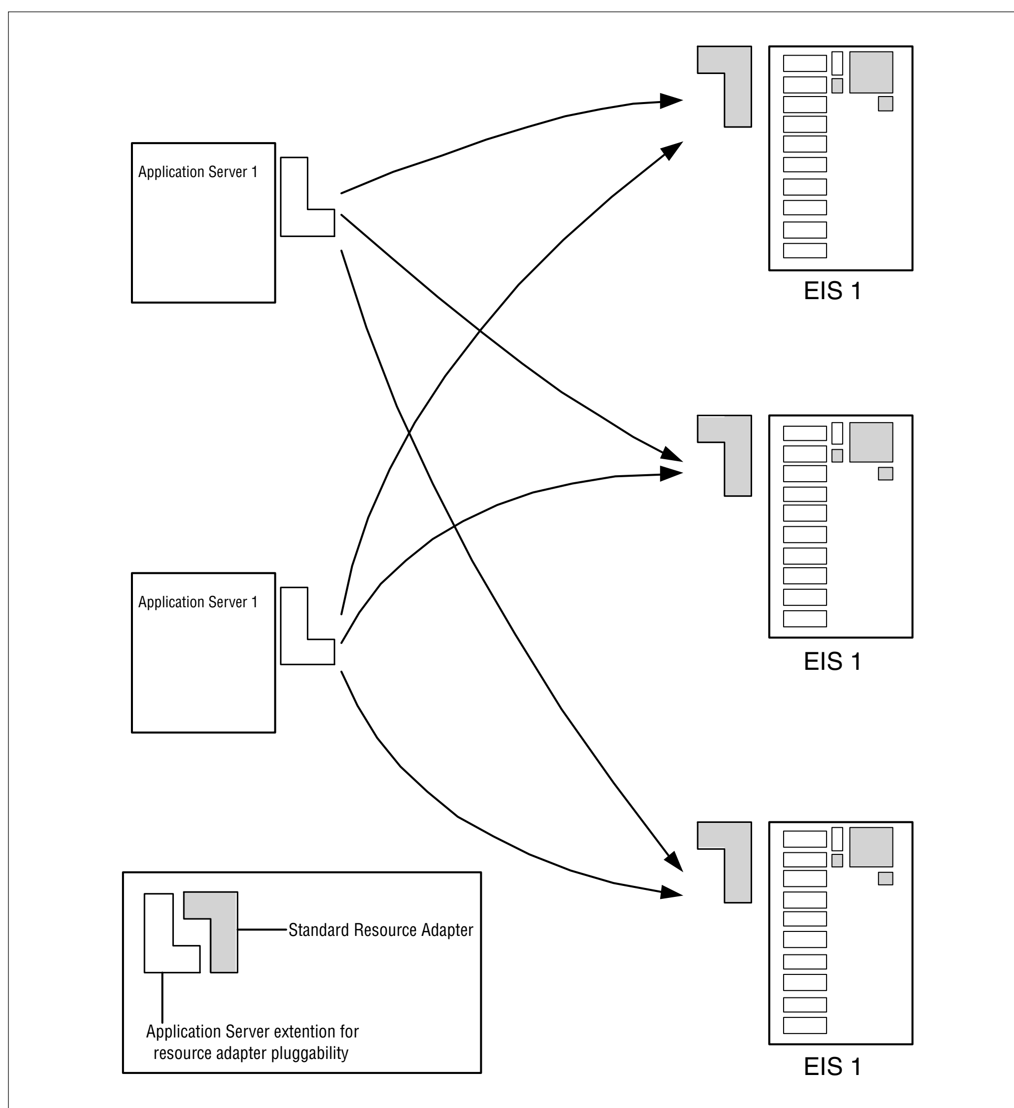


Figura 11.3 – *Installazione "pluggable" dei ResourceAdapter JCA.*



È bene comunque notare che l'implementazione delle API CCI non è mandatoria. Nel caso in cui non sia disponibile l'accesso via CCI, utilizzando direttamente l'interfaccia proprietaria del RA, si perderebbe l'indipendenza del codice dallo specifico EIS.

L'approccio JCA consente in generale di ridurre la complessità delle connessioni da gestire. Si veda ad esempio il caso di *n* application server che hanno la necessità di connettersi con *m*

EIS. Affrontando il problema con una soluzione proprietaria (senza JCA), si dovrebbero sviluppare $n \times m$ connessioni punto-punto.

Affrontando la medesima situazione con l'ausilio di JCA, il problema viene ridimensionato passando a una gestione di $m + n$ connessioni risolte mediante l'installazione del RA su ogni application server come mostrato nella fig. 11.3.

Gli elementi dell'architettura JCA

Vediamo nel dettaglio i componenti dell'architettura JCA.

Resource Adapter

Il RA è il driver software che permette al client (sia esso un componente J2EE come un EJB o l'application server stesso) di connettersi fisicamente alla risorsa EIS.

Un connettore "JCA compliant" deve implementare le interfacce di sistema (System Contracts) che gli consentono di interagire con le risorse gestite dall'application server (resource pool, transaction manager, sicurezza, e così via).

Il RA viene utilizzato all'interno dello spazio d'indirizzamento. Come si è visto, per agevolarne la gestione e l'installazione, i connettori sono "visti" dall'application server come dei componenti plugin. Per raggiungere questo livello di *system-level pluggability* l'architettura JCA definisce un set standard di "contratti" sia a livello di sistema (Application Server/RA), sia a livello EIS (RA/EIS).

Il RA può supportare diversi gradi di interazione con l'application server. Parlando ad esempio di supporto transazionale, un RA può fornire una gestione non transazionale, locale all'application server oppure distribuita XA-based.

Il RA è contenuto in un file di tipo *archive* (Resource Adapter Archive, RAR) composto dai file Java e dalle eventuali librerie native necessarie per accedere alla risorsa EIS. Un file `.rar` rappresenta a tutti gli effetti una risorsa J2EE e può essere installato singolarmente o all'interno di un file `.ear`.

System contract

Un adapter JCA interagisce con le infrastrutture del server J2EE tramite i cosiddetti system contracts. JCA specifica il modo in cui il sistema middleware deve collaborare con la risorsa EIS al fine di gestire i meccanismi di sistema. JCA definisce i contratti di Connection, Security e Transaction.

Connection Management

Determina le politiche per stabilire/chiudere le connessioni con le risorse di backend ottimizzando la gestione degli accessi mediante un Connection Pool gestito dall'application server. Permette inoltre di definire dei listener per gestire eventi associati alla gestione delle connessioni.

Transaction Management

Permette di gestire l'accesso transazionale verso le risorse EIS, definendo le interazioni tra la risorsa e il transaction manager.

Le transazioni possono richiedere la presenza di un transaction manager esterno oppure possono essere gestite internamente dal resource manager della risorsa. Questo contratto non è obbligatorio in quanto la specifica ammette RA non transazionali.

Security

Permette di gestire la Sicurezza per accedere all'EIS. Consente allo sviluppatore del RA di implementare/fornire i meccanismi per la gestione dell'autenticazione e dell'autorizzazione, ed eventualmente consente la gestione di un security environment custom per la comunicazione sicura tra il server J2EE e la risorsa EIS. Lo specifico meccanismo di sicurezza usato è indipendente dal meccanismo di sicurezza del sistema di backend.

JCA vs. Web Service

Tipicamente JCA viene messo in contrapposizione con la tecnologia Web Service. Nonostante ambedue gli standard operino nello scenario EAI, il paragone è sicuramente riduttivo.

Entrambi gli standard sono pensati per consentire la definizione di interfacce standard per generiche risorse enterprise. Mentre però JCA si orienta alle risorse di backend, la piattaforma Web Service ha uno scope maggiore ed è pensata per l'esposizione di servizi ad ampio spettro.

Sebbene anche i web service possano essere utilizzati per l'esposizione di EIS, i layer su cui operano naturalmente i due standard sono sostanzialmente differenti. Per l'integrazione di risorse di backend, lo standard JCA è sostanzialmente più ricco: non ha i limiti di protocollo HTTP, gestisce la transazionalità etc. D'altro canto è evidente come la piattaforma web service ponga minori complessità d'implementazione, non dipenda necessariamente dall'implementazione dei vendor (i RA devono essere reperiti sul mercato in quanto richiedono conoscenze troppo dettagliate dell'EIS) e soprattutto non sia supportata solamente dall'architettura Java, ma anche da quella Microsoft.

Anche se sul mercato i due standard sono posti in sostanziale concorrenza, in uno scenario Java è naturale pensare all'uso congiunto delle due tecnologie: web service utilizzati per l'esposizione dei servizi verso l'esterno, JCA utilizzato per la comunicazione interna con il backend.

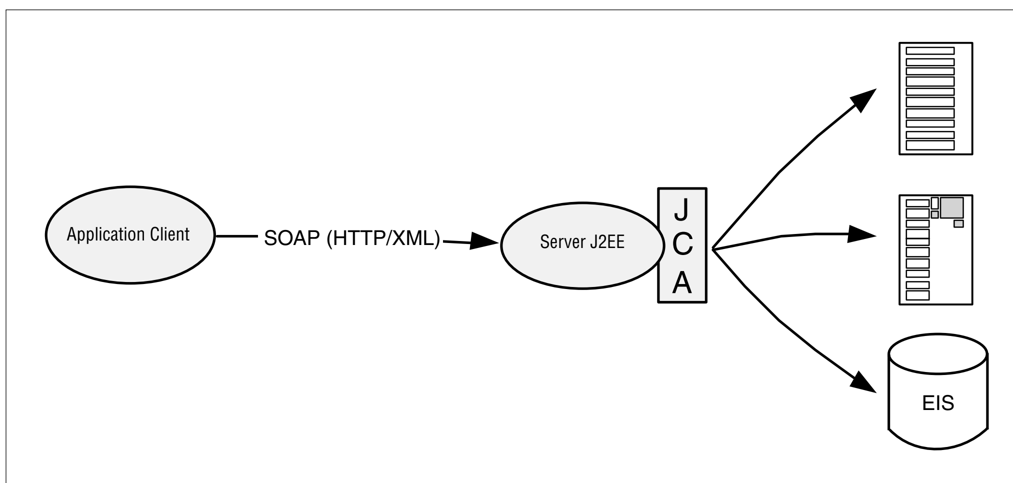
Il mercato

Sebbene JCA possa essere considerato un passo importante per il mondo Java e per il mercato EAI in genere, la versione attuale dello standard presenta alcuni limiti e non soddisfa tutti i requisiti legati a un reale progetto di integrazione.

Nonostante le notevoli promesse, l'impatto di JCA sul mercato potrebbe quindi essere limitato. Il peso che JCA assumerà negli anni a venire sarà, come per ogni proposta di standard, soprattutto dettato dal supporto e dalla fiducia che i principali player del mercato decideranno di accordare alla specifica Sun, soprattutto nell'eventualità che altre specifiche solo in parte alternative, come i web service, riescano a imporsi rapidamente sul mercato.

Ad oggi molti produttori già dichiarano ufficialmente di supportare la specifica JCA; tra questi spiccano BEA, IBM, IONA, Oracle e TIBCO. Le più recenti versioni degli application server sono JCA-compliant; tra questi BEA Weblogic, IBM WebSphere e OracleAS. Inoltre sul

Figura 11.4 – *JCA e Web Service.*



mercato sono già disponibili molti Resource Adapter per differenti EIS (tra questi SAP, CICS, IMS, Oracle, PeopleSoft). Per una lista completa, si veda [PROD].

Common Client Interface (CCI)

Per accedere da un'applicazione J2EE a un Resource Adapter (RA), l'architettura JCA mette a disposizione delle API denominate CCI (Common Client Interface).

CCI definisce un insieme di API per uniformare l'accesso a EIS da parte di applicazioni client. I client CCI possono essere componenti applicativi quali EJB o componenti d'integrazione come gli stessi integration broker.

CCI fornisce API di alto livello che permettono di astrarsi dai dettagli e dalle specificità di gestione delle differenti risorse EIS, operando rispetto a queste in modo analogo a JDBC rispetto ai DBMS, JMS rispetto ai sistemi di messaging o JNDI rispetto ai servizi di naming e directory.

L'analogia è ancora più stretta se si confronta JCA con JDBC; si può scherzosamente affermare che JCA è il "cugino di primo grado" di JDBC [BSTW].

Le API CCI astraggono lo sviluppatore dai dettagli di comunicazione con lo specifico EIS e definiscono uno standard per accedere a qualsiasi EIS non-relazionale. In modo similare, le API JDBC astraggono lo sviluppatore dai dettagli di comunicazione con lo specifico DBMS e definiscono uno standard per accedere a database relazionali.

Per creare una connessione con JDBC si utilizza `javax.sql.DataSource`

```
dataSource = (DataSource) ic.lookup("java:comp/env/jdbc/MyDataBase");
```

In JCA si utilizza javax.resource.cci.ConnectionFactory

```
factoryJCA = (ConnectionFactory) ic.lookup("java:comp/env/eis/MyHost");
```

Figura 11.5 – *JCA e le altre tecnologie Java.*

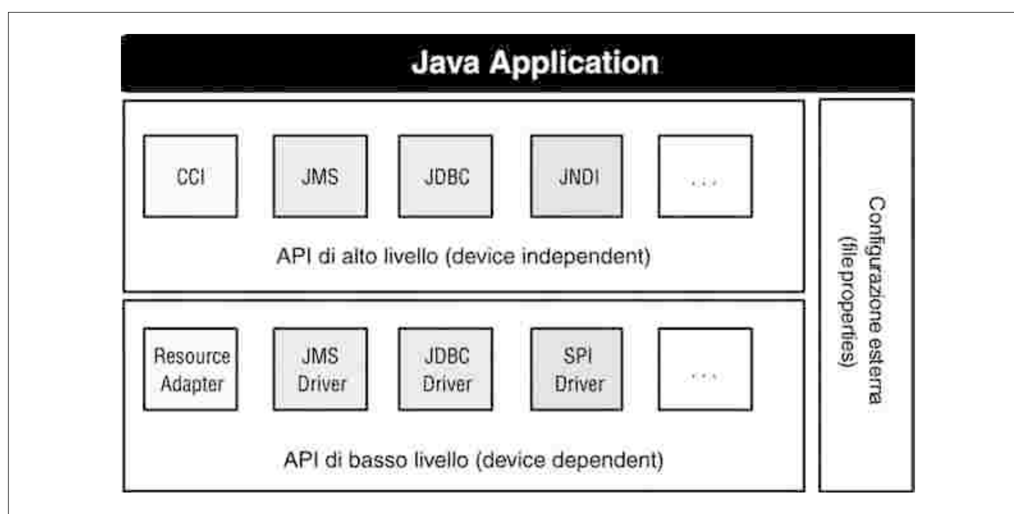
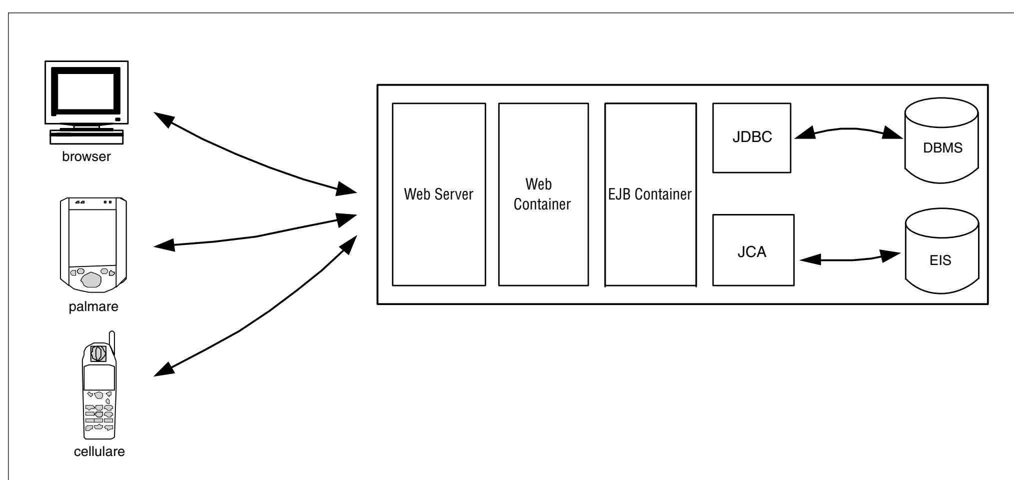


Figura 11.6 – *JCA e JDBC.*



Per ottenere una connessione con il DBMS in JDBC si utilizza il metodo getConnection dell'oggetto DataSource

```
java.sql.Connection con = dataSource.getConnection();
```

In JCA tutto ciò avviene in modo analogo, invocando il metodo getConnection sull'oggetto ConnectionFactory

```
javax.resource.cci.Connection con = cf.getConnection(. . .);
```

Come già detto l'implementazione delle API CCI non è mandatoria. Se l'accesso via CCI non è disponibile, è necessario utilizzare direttamente l'interfaccia proprietaria del RA; ovviamente l'utilizzo diretto di API proprietarie riduce i vantaggi di astrazione di JCA.

Panoramica API CCI

Le API CCI si dividono fondamentalmente in quattro aree:

- *Connection API*: per connettersi all'EIS;
- *Interaction API*: per eseguire comandi e invocare azioni sull'EIS;
- *Record/ResultSet API*: incapsulano il risultato dell'operazione avvenuta sull'EIS;
- *Metadata API*: che permettono la descrizione dei tipi di dati.

Affinchè un RA sia CCI-compliant deve implementare le intefacce che verranno illustrate tra poco. Le interfacce obbligatorie sono javax.resource.cci.Connection e javax.resource.cci.Interaction. Le API CCI sono costituite da sole interfacce, incluse nel package javax.resource.cci.

ConnectionFactory (javax.resource.cci.ConnectionFactory)

È l'interfaccia Factory responsabile della creazione delle connessioni con l'EIS. Il ConnectionFactory è pubblicato in un namespace JNDI e quindi reperibile mediante l'operazione JNDI di lookup.

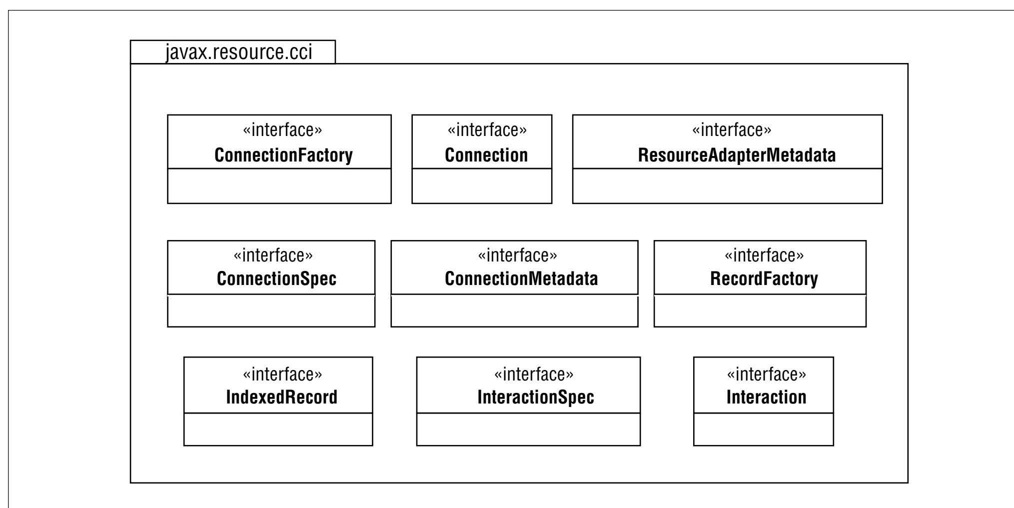
```
ConnectionFactory cf = (ConnectionFactory)
    context.lookup("java:comp/env/eis/CCI_MOKA_EIS");
```

Per ottenere la connessione con l'EIS, un componente J2EE deve invocare il metodo getConnection del ConnectionFactory.

Connection (javax.resource.cci.Connection)

Rappresenta la connessione con il sottosistema EIS. La Connection si ottiene invocando il metodo getConnection su un'istanza di ConnectionFactory.

Figura 11.7 – *Le interfacce javax.resource.cci.*




```
Connection con = cf.getConnection(spec);
```

Permette inoltre di creare un oggetto Interaction per interagire con la risorsa EIS.

```
Interaction ix = con.createInteraction();
```

Il metodo close permette di chiudere la connessione con l'EIS.

```
try { con.close();}
catch (ResourceException ex) {. . .}
```

ConnectionMetaData (javax.resource.cci.ConnectionMetaData)

Permette di specificare le informazioni sull'istanza EIS puntata dal connection reference. I metadata sono ottenibili invocando il metodo getMetaData sull'oggetto Connection.

```
ConnectionMetaData conMetaData =  con.getMetaData();
System.out.println("EISProductName="+conMetaData.getEISProductName());
System.out.println("ProductVersion="+conMetaData.getEISProductVersion());
System.out.println("UsertName ="+conMetaData.getUserName());
```

ConnectionSpec (javax.resource.cci.ConnectionSpec)

L'implementazione di questa interfaccia permette al client di comunicare al ConnectionFactory eventuali proprietà specifiche (*request-specific properties*) della risorsa EIS durante la richiesta

di connessione. Questa classe è di fatto un Java bean con proprietà, metodi di accesso (*mutators methods – setter*) e metodi di lettura (*accessors methods – getter*).

```
ConnectionSpec spec = new CciConnectionSpec(user, password);
Connection con = connectionFactory.getConnection(spec);
```

ResourceAdapterMetaData (javax.resource.cci.ResourceAdapterMetaData)

È l'interfaccia che fornisce informazioni sulle caratteristiche del Resource Adapter. È ottenibile invocando il metodo getMetaData sull'oggetto ConnectionFactory.

Tale invocazione quindi non implica che sia attiva una connessione con la risorsa EIS in quanto si limita a fornire informazioni descrittive e di utilizzo del RA.

```
ResourceAdapterMetaData raMetaData=cf.getMetaData();
System.out.println(raMetaData.getAdapterName());
System.out.println(raMetaData.getAdapterShortDescription());
System.out.println(raMetaData.getAdapterVendorName());
System.out.println(raMetaData.getAdapterVersion());
System.out.println(raMetaData.getInteractionSpecsSupported());
System.out.println(raMetaData.getSpecVersion());
System.out.println(raMetaData.supportsExecuteWithInputAndOutputRecord());
System.out.println(raMetaData.supportsExecuteWithInputRecordOnly());
System.out.println(raMetaData.supportsLocalTransactionDemarcation());
```

InteractionSpec (javax.resource.cci.InteractionSpec)

Contiene le eventuali proprietà necessarie per interagire con la risorsa EIS. È utilizzata dall'oggetto Interaction per l'esecuzione della specifica funzione della risorsa EIS.

Come ConnectionSpec, anche InteractionSpec è di fatto un JavaBean.

```
CciInteractionSpec iSpec = new CciInteractionSpec();
iSpec.setFunctionName("WRITE");
…
```

Interaction (javax.resource.cci.Interaction)

Permette di interagire con la risorsa EIS indicando la modalità d'accesso (scrittura/lettura). L'oggetto Interaction viene costruito a partire dall'oggetto Connection.

```
Interaction ix = con.createInteraction();
```

Il metodo close chiude l'Interaction corrente e rilascia tutte le risorse associate ma non la connessione con l'EIS. È buona pratica invocare in modo esplicito il metodo close alla fine di ogni interazione.

```
ix.close();
```

L'interazione con la risorsa EIS avviene mediante il metodo execute fornito con le firme riportate di seguito.

```
public boolean execute(InteractionSpec ispec,Record input,Record output) throws ResourceException
```

Esegue l'interazione con la risorsa EIS aggiornando il valore dell'output Record, restituendo true se l'operazione è andata a buon fine. Se l'interazione fallisce, il metodo solleva una javax.resource.ResourceException.

```
public Record execute(InteractionSpec ispec, Record input) throws ResourceException
```

Esegue l'interazione con la risorsa EIS ricevendo in ingresso un Record e restituendo come output un nuovo Record nel caso l'operazione abbia successo.

Record (javax.resource.cci.Record)

È l'interfaccia madre delle varie tipologie di Record. Rappresenta i dati di input e output necessari per eseguire le interazioni con l'EIS. Esempi di interfacce derivate sono:

- MappedRecord: record di coppie di chiave-valore basati sull'interfaccia java.util.Map;
- IndexedRecord: collezione ordinata e accessibile mediante indice;
- ResultSet: estende sia l'interfaccia Record, sia l'interfaccia java.sql.ResultSet per reperire dati in forma tabellare.

È inoltre possibile creare una propria tipologia di Record in funzione delle esigenze specifiche dell'EIS (per esempio, un Record per rappresentare un ordine di acquisto presso un sistema ERP).

RecordFactory (javax.resource.cci.RecordFactory)

È l'interfaccia factory responsabile della creazione d'istanze di Record. Così come il ConnectionFactory permette di avere oggetti Connection, il RecordFactory permette di ottenere oggetti Record. Mette a disposizione un metodo factory specifico per ogni tipologia di record (createIndexedRecord, createMappedRecord, createResultSet).

LocalTransaction (javax.resource.cci.LocalTransaction)

È l'interfaccia utilizzata per demarcare (begin, commit e rollback) le transazioni locali a un RA. La gestione della LocalTransaction è interna al RA e non coinvolge coordinamento esterno.

Un'istanza valida di LocalTransaction è ottenibile, se l'implementazione CCI lo supporta, invocando il metodo Connection.getLocalTransaction.

Come utilizzare un connettore

Dopo la panoramica sulle API CCI, vediamo come un client J2EE può accedere ad un RA. Le operazioni che devono essere effettuate sono riportate di seguito.

Importare il package javax.resource.CCI e la classe javax.resource.ResourceException:

```
import javax.resource.cci.*;
import javax.resource.ResourceException;
```

Effettuare, mediante JNDI, la lookup dell'oggetto ConnectionFactory del Resource Adapter:

```
Context ctx = new InitialContext();
Object obj = ctx.lookup("java:comp/env/eis/CCI_MOKA_EIS");
```

effettuando l'opportuno downcast

```
ConnectionFactory cf = (ConnectionFactory)obj;
```

Creare l'oggetto ConnectionSpec per indicare gli eventuali parametri specifici necessari a ottenere la conessione all'EIS:

```
ConnectionSpec spec = new CciConnectionSpec(<<param1>>,. . .,<<paramN>>);
```

Invocare sull'oggetto ConnectionFactory il metodo getConnection per ottenere l'apertura della connessione con il Resource Manager dell'EIS:

```
Connection con = cf.getConnection(spec);
```

L'oggetto Connection rappresenta la connessione con l'EIS e permette di interagire con la risorsa stessa. Invocare sull'oggetto Connection il metodo createInteraction per creare un'istanza Interaction. Mediante l'istanza ottenuta sarà possibile interagire con la risorsa EIS:

```
Interaction ix = con.createInteraction();
```

Per specificare eventuali parametri specifici per interagire con la risorsa EIS (ad esempio il nome di una tabella da accedere, il nome di una transazione, i parametri dell'operazione e così via) si utilizza l'oggetto InteractionSpec:

```
CciInteractionSpec iSpec = new CciInteractionSpec();
iSpec.setFunctionName(<<Operazione EIS da eseguire>>);
```

Creare RecordFactory invocando il metodo getRecordFactory sull'oggetto ConnectionFactory.

```
RecordFactory rf = cf.getRecordFactory();
```

Utilizzare l'oggetto RecordFactory per costruire gli appropriati record di lettura e/o scrittura. Tali record possono essere di input, output o sia di input che di output (record inout):

```
IndexedRecord iRec = rf.createIndexedRecord("InputRecord");
iRec.add(new Integer(userid));
iRec.add(username);

Record oRec = null; // Ouput Record
```

Eseguire le opportune operazioni sul data store invocando il metodo execute sull'oggetto Interaction, fornendo gli opportuni Record di input, di output o di inout:

```
Record oRec = ix.execute(iSpec, iRec);
Iterator iterator = ((IndexedRecord)oRec).iterator();
while(iterator.hasNext()) {
  Object obj = iterator.next();
  . . .
}
```

oppure

```
if(ix.execute(iSpec, iRec, oRec)) {
  // success
}
else {
  // failed
}
```

Conclusa la comunicazione con la risorsa EIS è necessario effettuare il codice di chiusura invocando sull'oggetto Connection il metodo close:

```
  con.close();
```

Se il RA supporta la gestione delle transazioni, prima di iniziare l'interazione con la risorsa EIS si deve demarcare l'inizio della transazione:

```
// ottengo un reference al transaction context
LocalTransaction transaction = con.getLocalTransaction();
// avvio della transazione
transaction.begin();
```

Conclusa l'operazione si deve gestire la commit se l'operazione ha avuto successo o la rollback in caso si siano rilevati errori:

```
try {
   ix.execute(iSpec, iRec);
   transaction.commit();
} catch (ResourceException e) {
   transaction.rollback();
}
// Close the connection
…
```

La tab. 11.1 riassume le interfacce CCI che il Resource Adapter deve implementare.

I System Contracts

I *system contracts* sono i contratti che si stipulano tra una risorsa J2EE esterna – nel nostro caso è il Resource Adapter – e i servizi della piattaforma J2EE ospitante (cioè l'Application Server). Tali contratti permettono di specificare le relative interazioni tra i due partecipanti.

Nella versione 1.0, JCA specifica mediante i contratti di Connection, Transaction e Security come il sistema middleware debba collaborare con la risorsa EIS al fine di gestire i meccanismi

Tabella 11.1 – *Interfacce CCI che devono essere implementate dal Resource Adapter.*

Interfaccia CCI implementata **javax.resource.cci.**	**Scopo**
ConnectionFactory	Permette di ottenere una connessione logica con l'EIS.
Connection	Rappresenta la connessione logica con l'EIS.
ConnectionMetaData	Incapsula i metadata della conessione CCI.
ConnectionSpec	Permette di specificare i parametri, vendor-dependent, della richiesta di conessione.
Interaction	Permette di interagire con la risorsa EIS.
InteractionSpec	Contiene le proprietà, vendor-dependent, specifiche della risorse.
RecordFactory	Permette di creare oggetti di tipo Record.
IndexedRecord	È una collezione ordinata di Record

di sistema. La versione 1.5 introduce quattro nuovi contratti: Lifecycle Management Contract, Work Management Contract, Transaction Inflow Contract e Message Inflow Contract.

Connection Management

Il Connection Management è il contratto che determina le politiche per stabilire/chiudere le connessioni con le risorse EIS ottimizzando la gestione degli accessi mediante un connection pool gestito dall'application server.

Permette anche di definire dei listener per gestire eventi associati alla gestione delle connessioni.

Transaction Management

Consente di gestire l'accesso transazionale verso le risorse EIS, definendo le interazioni tra la risorsa e il transaction manager. Le transazioni possono richiedere la presenza di un transaction manager esterno oppure possono essere gestite internamente dal resource manager della risorsa EIS stessa.

Questo contratto non è obbligatorio in quanto la specifica JCA ammette RA non transazionali.

Security

Permette di gestire la sicurezza per accedere all'EIS. Consente allo sviluppatore del RA di implementare/fornire i meccanismi per la gestione dell'autenticazione e dell'autorizzazione, ed eventualmente consente la personalizzazione della comunicazione sicura tra il server J2EE e la risorsa EIS.

Le classi coinvolte nella realizzazione di ogni contratto JCA si possono dividere in due "categorie": le classi *Managed* e le classi *Physical Connection*.

Le classi Managed interagiscono direttamente con l'application server ospitante, ad esempio permettono la gestione delle connessioni mediante connection pool o di interagire con il transaction manager per la gestione delle transazioni.

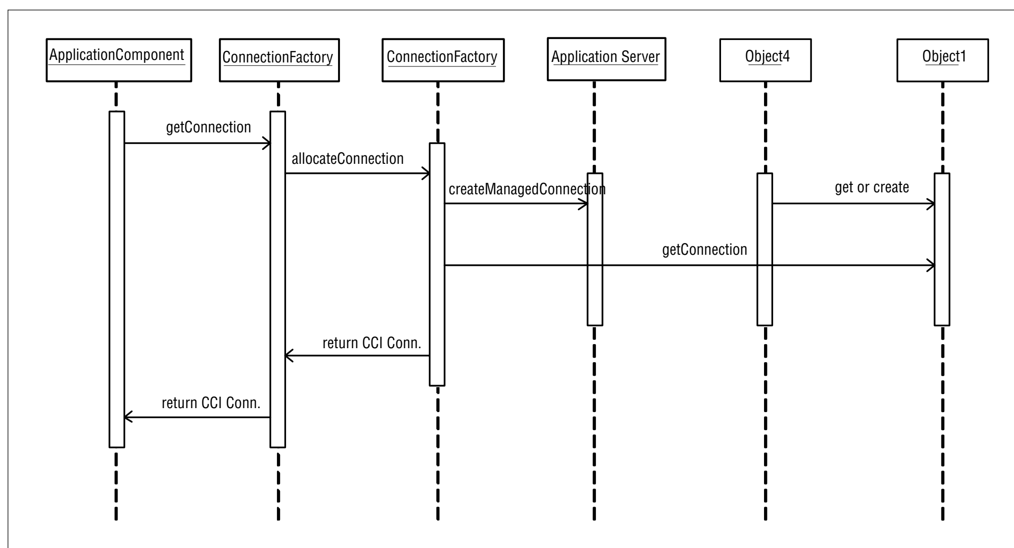
L'altra categoria comprende le classi che gestiscono direttamente l'interazione diretta con l'EIS.

Connection Management

Il Resource Adaptor deve essere in grado di agire come un factory di connessioni (pattern Factory) fornendo un'implementazione delle interfacce javax.resource.spi.ManagedConnectionFactory e javax.resource.spi.ManagedConnection, al fine di fornire rispettivamente un costruttore di connessioni e una classe che gestisca il collegamento fisico della risorsa EIS. Ogni Resource Adapter deve almeno fornire l'implementazione delle seguenti tre interfacce SPI:

- ManagedConnectionFactory: agisce da factory delle connessioni;
- ManagedConnection: rappresenta la connessione fisica con l'EIS;
- ManagedConnectionMetaData: fornisce le informazioni dell'EIS associato alla relativa istanza ManagedConnection;

Figura 11.8 – *Sequence diagram delle interazioni per ottenere una connessione con il RA.*



Tali interfacce devono essere obbligatoriamente implementate per poter realizzare un Resource Adapter JCA. Le API Service Provider Interface (SPI) appartenenti al package javax.resource.spi contengono le classi per gestire la connessione con la risorsa EIS fornendo un framework per l'invocazione del servizio. Nel prossimo paragrafo si entrerà nel dettaglio delle implementazioni di tali classi parlando del MokaConnector.

Nel sequence diagram di fig. 11.8 si cerca di dare un'idea complessiva di come il client JCA sia in grado di ottenere una connessione con l'EIS. Il diagramma di sequenza mette in rilievo anche l'interazione tra le classi Managed e le classi CCI. Si ricorda che, mentre le classi Managed sono obbligatorie per costruire un RA, le classi CCI si hanno solo nel caso in cui il RA sia CCI-compliant; quindi le classi Managed restituiscono al client JCA o classi che implementano interfacce CCI oppure classi proprietarie.

Il client JCA invoca il metodo getConnection() del connection factory (nel caso del MokaConnector è la classe MokaCciConnectionFactory), il quale si limita a invocare il metodo allocateConnection() dell'oggetto ConnectionManager dell'application server.

È l'application server che interagisce con le risorse Managed. Infatti come prima operazione l'AS invoca il metodo createManagedConnection() della classe di tipo managed ConnectionFactory. A questo punto viene creato un nuovo oggetto di classe ManagedConnection (la connessione con l'EIS), oppure viene prelevato il primo disponibile dal connection pool. Mediante l'invocazione del metodo getConnection() della classe ManagedConnection viene restituito al client il reference della connessione richiesta.

Un Resource Adapter può inoltre fornire la classe che implementa l'interfaccia javax.resource.spi.ConnectionRequestInfo e permette di descrivere i dati specifici della richiesta di

connessione all'EIS. Il ConnectionRequestInfo viene passato come parametro al metodo allocateConnection() del ConnectionManager durante la fase di reperimento della connessione conl'EIS. Il sequence diagram di fig. 11.8 ha mostrato le interazioni tra il client JCA, le classi Managed, le classi CCI e l'AS al fine di ottenere una connessione con l'EIS.

Vediamo di descrivere con maggior dettaglio il ruolo di ognuna della classi che permettono la gestione delle connessioni del RA.

ManagedConnectionFactory

È la classe che agisce da entry-point per la comunicazione tra l'application server ed il RA.

Permette la creazione di connessioni con l'EIS o interagendo con il connection pool dell'AS, o creandone delle nuove a seconda della politica di gestione delle connessioni attuata.

È importante capire che, come un'applicazione J2EE utilizza il ConnectionFactory

```
Context ctx = new InitialContext();
Object obj = ctx.lookup("java:comp/env/eis/CCI_MOKA_EIS");
ConnectionFactory cf = (ConnectionFactory)obj;
Connection con = cf.getConnection(connectionSpec);
```

così l'application server utilizza il ManagedConnectionFactory per creare oggetti ManagedConnection, come rappresentato nel precedente sequence diagram.

ManagedConnection

È la classe che incapsula la connessione fisica con la risorsa EIS, mette a disposizione il metodo getConnection() che permette di ottenere un handle alla connessione fisica della risorsa EIS.

```
public java.lang.Object getConnection(javax.security.auth.Subject subject,
      ConnectionRequestInfo cxRequestInfo)  throws ResourceException
```

Sempre mediante l'implementazione dell'interfaccia ManagedConnection è possibile accedere alle interfacce javax.resource.spi.LocalTransaction e javax.transaction.xa.XAResource per la gestione delle transazioni del RA.

ManagedConnectionMetaData

Invocando il metodo getMetaData() vengono restituite le informazioni descrittive della connessione incapuslate in un oggetto d'interfaccia ManagedConnectionMetaData.

Tale classe, di fatto, agisce da wrapper per gli specifici metadata della risorsa EIS.

```
public class MokaManagedConnectionMetaData implements javax.resource.spi.ManagedConnectionMetaData {
```

Le altre interfacce SPI [SPI], ad esempio la ConnectionEventListener (fornisce un meccanismo di callback a eventi per permettere di ricevere notifiche dal ManagedConnection), la ConnectionManager (fornisce un hook per permettere agli adapter di inviare richieste di connes-

sione all'application server), la ConnectionrequestInfo, e altre, non devono obbligatoriamente essere implementate dallo sviluppatore del RA.

Transaction management

Le interfacce del transaction management forniscono un framework che l'application server utilizza per gestire le transazioni della risorsa EIS. Le transazioni possono essere sia di tipo locale che distribuito (XA). Un Resource Adapter può non supportare le transazioni o essere in grado di gestire quelle locali, quelle distribuite oppure entrambe. Quindi le interfacce che un RA deve implementare dipendono strettamente dal livello di transazione che il connettore stesso è in grado di fornire. È la classe ManagedConnection che, oltre a fornire l'handle della connessione alla risorsa EIS, permette di ottenere le interfacce per la gestione del transaction contract.

Il metodo getLocaltransaction() provvede a fornire il supporto delle transazioni in ambito locale restituendo un oggetto che implementa l'interfaccia javax.resource.spi.LocalTransaction.

L'oggetto LocalTransaction permette la gestione della transazione internamente al Resource Manager senza necessitare di un Transaction Manager fornito dall'AS.

```
public javax.resource.spi.LocalTransaction.LocalTransaction getLocalTransaction()
      throws ResourceException{
```

Nel caso le transazioni locali non fossero supportate dal Resource Adapter, tale metodo deve lanciare l'eccezione ResourceException.

Il metodo getXAResource() permette di ottenere un oggetto che implementa l'interfaccia javax.transaction.xa.XAResource.

```
public javax.transaction.xa.XAResource getXAResource () throws javax.resource.ResourceException { … }
```

L'oggetto XAResource definisce il contratto tra il Resource Adapter e il Transaction Manager per la gestione delle transazioni in un contesto transazionale distribuito. Nel caso le transazioni distribuite non fossero supportate dal Resource Adapter, tale metodo deve lanciare l'eccezione ResourceException. Nel caso di un RA CCI-compliant, sarà possibile ottenere un'istanza LocalTransaction valida invocando il metodo Connection.getLocalTransaction() per gestire la transaction-demarcation mediante i metodi begin, commit e rollback.

Security Management

Il Security Contract permette di implementare un accesso sicuro alle risorse EIS da parte dell'applicazione client J2EE. È possibile integrare questro contratto, all'interno nelle interfacce del Connection Management con JAAS (Java Authentication and Authorization Service).

JAAS sono le API che Sun fornisce per scopi di autenticazione (autenticare il soggetto che sta eseguendo una certa porzione di codice Java) e di autorizzazione (autorizzare l'utente assicurarandosi che abbia i giusti permessi per eseguire determinate azioni).

L'autenticazione in JAAS avviene in un modo *pluggable*, cioè le applicazioni – seguendo l'ormai consolidata e affermata "filosofia Java", vedi JDBC, JMS, JNDI e JCA – rimangono indipendenti dal livello tecnologico d'autenticazione utilizzato.

Mediante la classe javax.security.auth.login.LoginContext il codice applicativo viene disaccoppiato dall'effettiva implementazione JAAS-compliant che verrà utilizzata per le operazioni di sicurezza.

Le classi e interfacce di JAAS si possono suddividere in 3 categorie: quelle comuni (Subject, Principal, Credential), quelle di autenticazione (LoginContext, LoginModule, CallbackHandler, Callback) e infine quelle di autorizzazione (Policy, AuthPermission, PrivateCredentialPermission).

Le interfacce JAAS che vengono utilizzate nel RA sono illustrate nei paragrafi seguenti.

javax.security.auth.Subject

Un oggetto di classe Subject rappresenta un gruppo di informazioni correlate ad una singola identità o a una persona. Include l'identità del soggetto denominato Principal e gli attributi di sicurezza chiamati Credentials. Un oggetto Subject è passato al RA dall'application server per le opportune operazioni di autenticazione e autorizzazione.

java.security.Principal

L'interfaccia Principal permette di rappresentare un'identità nel contesto di sicurezza utilizzato.

Oltre a queste classi JAAS, il security contract JCA include la classe final (non estendibile) javax.resource.spi.security.PasswordCredential per facilitare la comunicazione sicura tra AS e la risorsa EIS.

javax.resource.spi.security.PasswordCredential

Tale classe incapsula username e password passate dall'AS al RA. Oltre a definire i metodi get per accedere alle proprietà username e passsword, definisce anche i metodi get/setManagedConnectionFactory. Questo permette di associare all'oggetto PasswordCredentials un'istanza ManagedConnectionFactory per la quale username e password sono state assegnate dall'AS.

Figura 11.9 – *Le interfacce javax.resource.spi.*

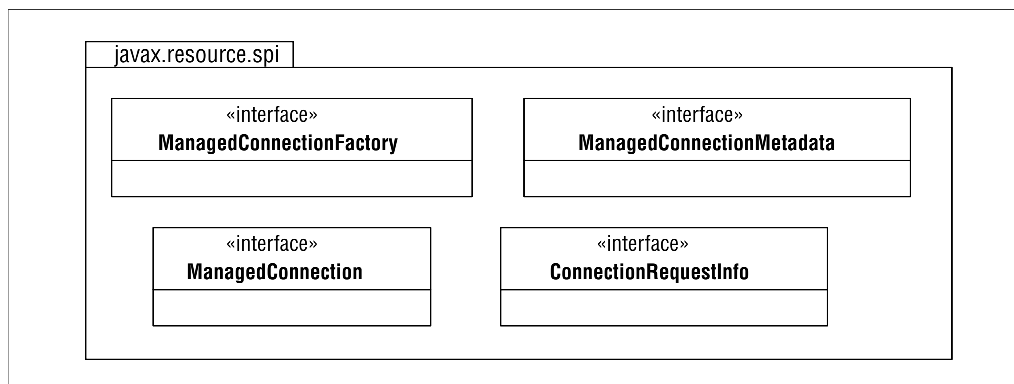


Tabella 11.2 – *Interfacce SPI che devono essere implementate da un Resource Adapter.*

Interfaccia SPI implementata javax.resource.spi	Scopo
ManagedConnectionFactory	Permette di ottenere una connessione fisica con l'EIS.
ManagedConnection	Rappresenta la connessione fisica con l'EIS.
ManagedConnectionMetaData	Incapsula i metadata della conessione fisica con l'EIS.
ConnectionRequestInfo	Incapsula le eventuali informazioni necessarie per richiedere una connessione all'EIS.
Nessuna	Gestione base della sicurezza.

JAAS e le classi del Connection Management

Le classi JAAS e la classe PasswordCredential sono utilizzate nelle interfacce JCA che gestiscono il Connection Management.

Le interfacce del Connection Management che partecipano all'implementazione della sicurezza del RA sono ConnectionFactory, ManagedConnectionFactory e ManagedConnection, a seconda di chi ha in carico il processo di login e a che livello è coinvolta la risorsa EIS interfacciata.

Nella tab. 11.2 sono riepilogate le interfacce SPI che un Resource Adapter deve implementare.

Dalla tabella si vede come tutte le classi si riferiscano al Connection Management ad eccezione della classe MokaConnectorSecurity che partecipa al Security Management. Il Transaction Contract non è gestito per l'intrinseca non-transazionalità della risorsa file.

Ogni RA deve obbligatoriamente fornire l'implementazione delle tre interfacce SPI: ManagedConnectionFactory, ManagedConnection e ManagedConnectionMetaData.

Il deploy di un connettore

Conclusa la panoramica delle API JCA, vediamo ora come effettuare deploy ed esecuzione.

Il file ra.xml

Ogni RA è "accompagnato" da un file descrittore di deploy di nome ra.xml posizionato sotto la directory META-INF (META-INF/ra.xml). Il deployment descriptor contiene diverse informazioni.

- Informazioni generiche e descrittive del RA:

```
<display-name>File System Resource Adapter</display-name>
<vendor-name>MokaByte S.R.L</vendor-name>
<spec-version>1.0</spec-version>
```

```
<version>1.0</version>
<eis-type>FileSystem</eis-type>
<license>
  <description>Mokabyte simple example</description>
  <license-required>false</license-required>
</license>
```

- Interfacce e relative classi concrete per accedere al connettore:

```
<managedconnectionfactory-class>
  it.mokabyte.mokajca.ra.MokaManagedConnectionFactory
</managedconnectionfactory-class>
<connectionfactory-interface>
  javax.resource.cci.ConnectionFactory</connectionfactory-interface>
<connectionfactory-impl-class>
  it.mokabyte.mokajca.ra.cci.MokaCciConnectionFactory
</connectionfactory-impl-class>
<connection-interface>
  javax.resource.cci.Connection
</connection-interface>
<connection-impl-class>
  it.mokabyte.mokajca.ra.cci.MokaCciConnection
</connection-impl-class>
```

- Supporto transazionale:

```
<transaction-support>NoTransaction</transaction-support>
```

- Informazioni di autenticazione (<authentication-mechanism>).
- Permessi di sicurezza (<security-permission>).
- Proprietà di configurazione:

```
<config-property>
  <config-property-name>FileName</config-property-name>
  <config-property-type>java.lang.String</config-property-type>
  <config-property-value>/moka_jca/eis/prova.txt</config-property-value>
</config-property>
```

Contiene quindi le informazioni necessarie per effettuare sia il deploy che la configurazione del RA per un AS JCA-enabled.

Deploy del Resource Adapter

Per effettuare il deploy del RA è necessario archiviare tutti i file in un JAR di estensione .rar. Il RA risulta quindi costituito da classi Java che implementano le API CCI, le classi del RA, eventuali driver nativi e il deployment descriptor XML. Questi file sono tutti archiviati nel medesimo file JAR che prende il nome di Resource ARchive file (RAR); un file RAR è l'unità di distribuzione del RA.

Una volta creato il file RAR è possibile effettuare l'operazione di deploy del RA in un server J2EE, analogamente a quanto accade per i .jar EJB, i file .war o gli stessi file .ear. La procedura di deploy è vendor-dependent, cioè varia a secondo dell'AS che si utilizza.

L'EJB di test utilizza il MokaConnector esplicitando come resource-reference il nome logico ra/CCI_MOKA_EIS nel file ejb-jar.xml:

```
<ejb-jar>
  <enterprise-beans>
    <session>
    <ejb-name>MokaEjbTestRA</ejb-name>
    <home>it.mokabyte.mokajca.ejb.TesterHome</home>
    <remote>it.mokabyte.mokajca.ejb.Tester</remote>
    <ejb-class>it.mokabyte.mokajca.ejb.TesterBean</ejb-class>
    <session-type>Stateful</session-type>
    <transaction-type>Container</transaction-type>
      <resource-ref>
      <res-ref-name>ra/CCI_MOKA_EIS</res-ref-name>
      <res-type>it.mokabyte.mokajca.ra.cci.MokaCciRecordFactory</res-type>
      <res-auth>Container</res-auth>
      </resource-ref>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Conclusioni

JCA (Java Connector Architecture) è un importante tentativo di standardizzare una parte del mercato dei prodotti di EAI (Enterprise Application Integration), attualmente dominato da soluzioni proprietarie. In questo capitolo abbiamo sommariamente presentato il contesto EAI in cui si inserisce la specifica JCA e abbiamo effettuato una panoramica sulla tecnologia dei Java Connector.

Riferimenti bibliografici

[JCA]
http://java.sun.com/j2ee/connector/

[SPEC]
J2EE Connector Architecture Specification
http://java.sun.com/j2ee/download.html#connectorspec

[CCI1]
BETH STEARNS, *Using the J2EETM Connector Architecture Common Client Interface*, aprile 2001
http://java.sun.com

[BSTW]
DAVID MARKS, *J2EE Connector Architecture Brings Business Systems to the web*,
http://developer.java.sun.com/developer/technicalArticles/J2EE/connectorclient/resourceadapter.html

[JW]
DIRK REINSHAGEN, *Connect the enterprise with the JCA*, JavaWorld

[JROD1]
JENNIFER RODONI, *The Java 2 Enterprise Edition Connector Architecture's Resource Adapter*
http://developer.java.sun.com/developer/technicalArticles/J2EE/connectorclient/resourceadapter.html

[JROD2]
JENNIFER RODONI, *What's New in the J2EE Connector Architecture 1.5*
http://developer.java.sun.com/developer/technicalArticles/J2EE/connectorarch1_5

[JTUT]
J2EE Tutorial
http://java.sun.com/j2ee/tutorial/download.html

[PROD]
JCA Product Information
http://java.sun.com/j2ee/connector/products.html

Capitolo 12

JMS

STEFANO ROSSINI

JMS è l'acronimo di Java Message Service e indica un'insieme di API che permettono lo scambio di messaggi tra applicazioni Java. Prima di descrivere in dettaglio JMS, è bene inquadrare il contesto del suo utilizzo; in particolare bisogna capire cosa si intende per "messaggio", o meglio, cosa si intende per *messaging system*.

MOM (Message Oriented Middleware)

Con il termine *messaging* si definisce un meccanismo che permette la comunicazione asincrona tra client (detti *peer-element*) mediante scambio di messaggi caratterizzato da uno scarso grado di accoppiamento.

Data questa definizione si può considerare messaging system un qualsiasi sistema che scambi pacchetti TCP/IP tra programmi client.

Piuttosto che creare un meccanismo di messaging specifico per ogni situazione che richieda lo scambio di dati, diversi produttori hanno creato un sistema che permette di scambiare messaggi tra applicazioni diverse in modo generico. Questi sistemi sono chiamati Message Oriented Middleware (MOM).

Tramite l'infrastruttura middleware del MOM (fig.12.1), processi residenti su macchine diverse possono interagire tra loro mediante l'invio e ricezione di messaggi tipicamente asincroni. È importante notare che si è in presenza di un sistema di messaging peer-to-peer dove i client interagiscono tra loro non in modo diretto bensì mediante un messaging server, che fa le veci del "postino": riceve i messaggi dai mittenti ("produttori") e li recapita ai relativi destinatari ("consumatori").

Grazie alla mediazione del messaging server, i client risultano essere disaccoppiati, cioè non devono sapere nulla l'uno dell'altro.

Il disaccoppiamento nei MOM è una caratteristica molto importante ed è maggiore rispetto ai sistemi middleware basati su architettura ORB (Object Request Broker). Si pensi ad esem-

Figura 12.1 – *Schematizzazione di un sistema MOM.*

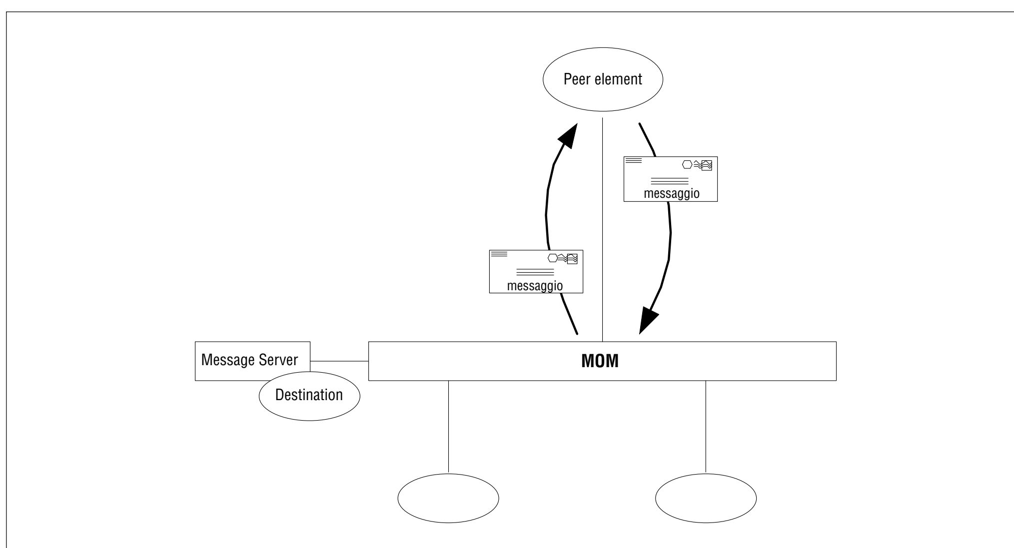
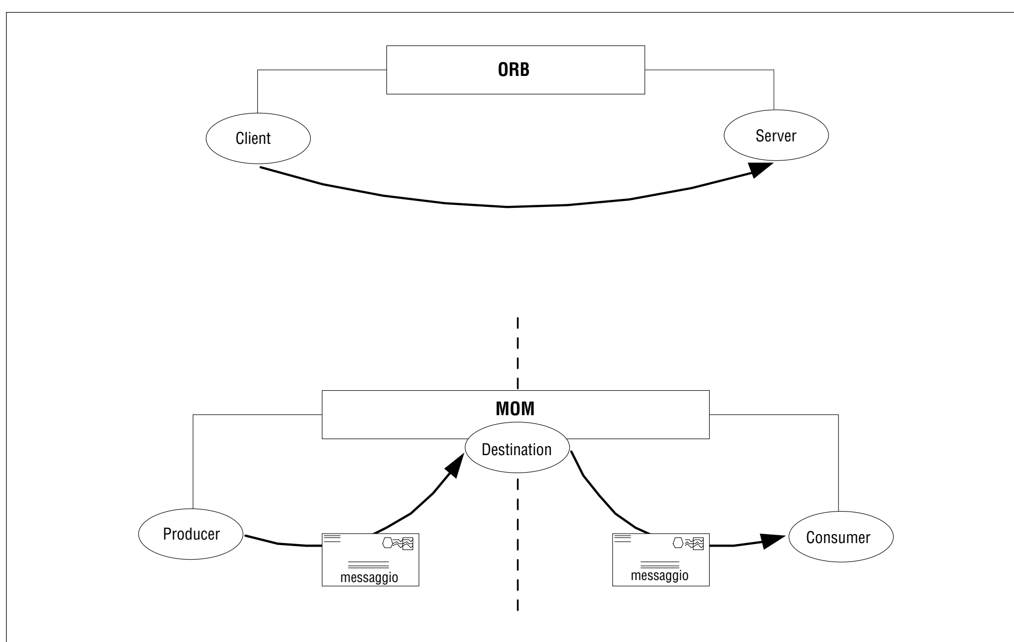


Figura 12.2 – *Il disaccoppiamento in un modello di messaging è maggiore rispetto a un modello RPC.*



pio a RMI o CORBA: per poter comunicare con un oggetto remoto, un'applicazione deve essere a conoscenza dei suoi metodi, in modo da poterli invocare (fig.12.2).

Nei sistemi MOM, i client non interagiscono direttamente tra di loro, e né il produttore né il ricevitore devono essere a conoscenza l'uno dell'altro. L'unica cosa che entrambi devono sapere è il formato del messaggio e la "casella di posta" (*destination*) da usare.

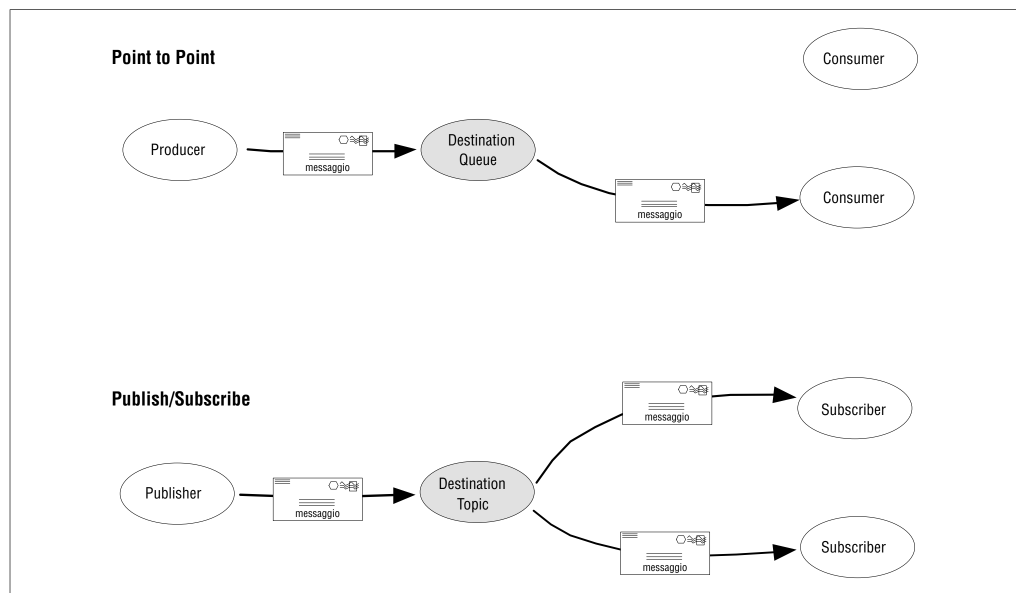
Altra importante caratteristica offerta dai sistemi MOM è il concetto di Guaranteed Message Delivery (GMD) che garantisce la consegna del messaggio. Al fine di prevenire una perdita di informazioni in caso di crash del message server, i messaggi devono essere resi persistenti (per esempio mediante JDBC) prima di essere recapitati ai consumatori.

Modelli di messaging

Dopo questa breve introduzione sul MOM vediamo i suoi due diversi modelli di messaging (fig. 12.3) con i quali è possibile scambiarsi i messaggi: *point-to-point* e *publish and subscribe*. Il modello di messaging point-to-point (PTP) è caratterizzato dal fatto che più produttori e più consumatori possono condividere la stessa destinazione (*queue*) e ogni messaggio ha un solo consumatore; questo permette di fatto una comunicazione uno-a-uno.

Il modello publish/subscribe (pub/sub) permette una comunicazione uno-a-molti dal momento che ogni messaggio inserito nella destinazione (*topic*) può essere inoltrato a tutti i destinatari che si sono dichiarati (*subscription*) interessati.

Figura 12.3 – *I modelli JMS.*



Point-to-point

Nel modello PTP un client può mandare uno o più messaggi a un altro client. La comunicazione non avviene in modo diretto bensì condividendo una stessa destinazione verso cui si inviano e da cui si prelevano i messaggi.

La destinazione nel modello point-to-point prende il nome di queue ("coda"). Più produttori e più consumatori possono condividere la stessa coda ma ogni messaggio ha un solo consumatore; questo permette di fatto una comunicazione uno a uno.

Non ci sono dipendenze temporali tra *sender* e *receiver* del messaggio e quindi il ricevente può ricevere il messaggio anche se non era in ascolto sulla coda al momento dell'invio. Il messaggio viene tolto dalla coda una volta che il ricevente l'ha prelevato confermando (*acknowledge*) la ricezione al produttore.

L'utilizzo del messaging PTP è da utilizzare nel caso si abbia l'esigenza che ogni messaggio spedito sia ricevuto da un solo consumatore che ne confermi l'avvenuta ricezione.

Publish/Subscribe

Nel modello publish/subscribe (detto anche sistema di messaging *event-driven*), i client inviano messaggi a un *topic*, una coda "speciale" in grado di inoltrare i messaggi a più destinatari.

Il sistema si prende carico di distribuire i messaggi inviati (*publish*) da più *publishers*: tali messaggi sono automaticamente ricevuti da tutti i client che hanno effettuato la sottoscrizione (*subscribe*), cioè che si sono dichiarati interessati alla ricezione.

Il modello *pub/sub* permette quindi una comunicazione uno-a-molti visto che ogni messaggio può avere più consumatori. È il JMS provider che si occupa di recapitare una copia di ogni messaggio pubblicato agli appropriati destinatari. Il messaggio una volta consumato da tutti i subscriber interessati viene tolto dal topic.

I client possono essere dinamicamente publisher oppure subscriber o anche entrambi. Esiste una dipendenza temporale tra publisher e subscriber, visto che il receiver può "consumare" il messaggio solo dopo essersi "sottoscritto" (*subscription*) al topic d'interesse, una volta che il client abbia effettivamente inviato il messaggio. Inoltre il receiver deve rimanere attivo, se vuole continuare a ricevere.

Le API JMS rilassano tale vincolo temporale mediante il concetto di *durable subscriptions*. Le durable subscriptions infatti permettono di ricevere i messaggi anche se i subscribers non erano in ascolto al momento dell'invio del messaggio.

L'utilizzo di messaging publish/subscribe è da preferire quando si ha la necessità che ogni messaggio sia ricevuto da più consumatori. Le destinazioni sono accessibili in modo concorrente e si possono definire anche dei filtri per esprimere le condizioni da rispettare affinché un messaggio venga recapitato a una certa destinazione.

Che cosa è JMS?

Java Message Service (JMS) è un'insieme di API Java che permette di creare, spedire, ricevere e leggere messaggi. JMS è stato sviluppato da Sun insieme ai maggior produttori di sistemi

MOM, per definire un'interfaccia comune indipendente dalla specifica implementazione del sistema di messaging, in modo analogo a quanto avviene con JDBC, JNDI, JCA e così via. L'applicazione Java JMS risulta così essere indipendente, oltre che dal sistema operativo, anche dalla specifica implementazione del sistema di messaging.

Mediante JMS è quindi possibile interagire con sistemi MOM esistenti quali MQSeries di IBM, TIBCO Rendevous, Fiorano MQ, Microsoft MQ, Progress SONIC MQ, BEA weblogic Server, ExoLab OPEN JMS, Softwired iBus.

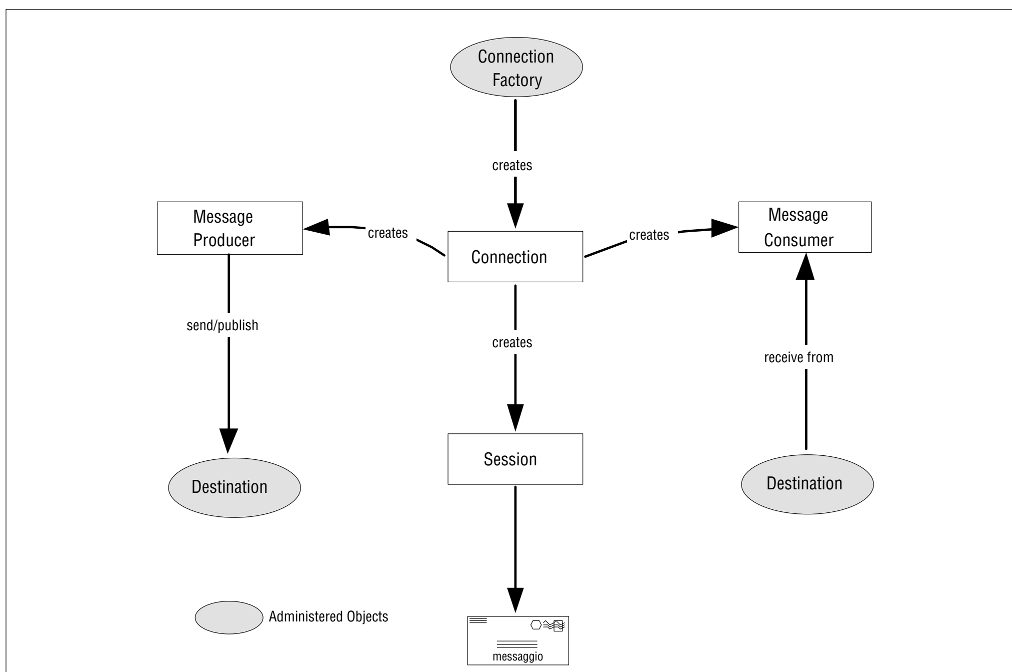
Si capisce come lo scarso disaccoppiamento rispetto ai sistemi RPC, l'intrinseca asincronia delle operazioni di messaging, la sua integrazione in ambito J2EE (i Message Driven Bean) e l'indipendenza dello specifico provider rendono JMS una tecnologia potente, flessibile e fondamentale soprattutto per interazioni business-to-business (B2B) e integrazioni di sistemi eterogenei in ambito EAI (Enterprise Application Integration).

Gli attori JMS

In un'applicazione JMS gli attori coinvolti sono `ConnectionFactory`, `Destination`, `Connection`, `Session`, `MessageProducer`, `Message` e `MessageConsumer`, come rappresentato in fig. 12.4.

Di seguito viene descritto brevemente il compito di ciascun attore JMS.

Figura 12.4 – *Gli attori JMS.*



Administered objects

Le Destination (queue/topic) e le ConnectionFactory sono dette *administered objects* perché incapsulano le specifiche implementazioni dei provider JMS. Gli administered objects sono *vendor-dependent*, cioè la loro specifica implementazione varia da provider a provider e per questo motivo non sono gestiti all'interno del programma Java.

Le interfacce JMS permettono allo sviluppatore Java di astrarsi dai dettagli implementativi della specifica versione di MOM, permettendo il riutilizzo del codice al variare dell'implementazione JMS, rispettando appieno la filosofia Java. La creazione di ConnectionFactory e di Destination è compito dell'amministratore JMS che deve inserirli all'interno di un contesto JNDI in un opportuno namespace in modo che possano essere recuperati da qualsiasi client mediante le normali API JNDI.

Per inviare messaggi (sender/publisher) o per riceverli (receiver/subscriber), un'applicazione JMS client utilizza i servizi JNDI per ottenere un oggetto di tipo ConnectionFactory e uno o più oggetti di tipo Destination (Queue o Topic).

ConnectionFactory

È l'administered object utilizzato dal client per ottenere una connessione con il message server. Nel caso point-to-point si utilizza l'interfaccia QueueConnectionFactory:

```
Context ctx = new InitialContext();
QueueConnectionFactory queueConnectionFactory = (QueueConnectionFactory)ctx.lookup("MyQueueConnectionFactory");
```

mentre nel caso di publish/subscribe si utilizza l'interfaccia TopicConnectionFactory:

```
TopicConnectionFactory topicConnectionFactory
= (TopicConnectionFactory)ctx.lookup("MyTopicConnectionFactory");
```

Destination

È l'administered object che rappresenta l'astrazione di una particolare destinazione. Anche in questo caso il client identifica la destinazione mediante l'utilizzo delle API JNDI. Nel caso point-to-point, la destinazione è rappresentata dall'interfaccia Queue

```
Queue queue = (Queue) jndiContext.lookup("MyQueue");
```

mentre, nel caso di publish/subscribe, si usa l'interfaccia Topic

```
Topic topic = (Topic) jndiContext.lookup("MyTopic");
```

Connection

Rappresenta l'astrazione di una connessione attiva con un particolare JMS provider e si ottiene dall'oggetto ConnectionFactory. Nel caso point-to-point, si ottiene un reference d'interfaccia QueueConnection invocando il metodo create QueueConnection sull'oggetto Connection Factory:

```
QueueConnection queueConnection = queueConnectionFactory.createQueueConnection();
```

Analogamente, nel caso di publish/subscribe, si ottiene un reference d'interfaccia TopicConnection invocando il metodo createTopicConnection() sull'oggetto ConnectionFactory:

```
TopicConnection topicConnection = topicConnectionFactory.createTopicConnection();
```

Session

L'interfaccia Session si ottiene dall'oggetto Connection e rappresenta il canale di comunicazione con una destinazione: permette la creazione sia di produttori che di consumatori di messaggi.

La creazione della sessione permette di specificare il controllo della transazione e la modalità di acknowledge. Nel caso point-to-point, la Session è identificata mediante l'interfaccia QueueSession:

```
QueueSession queueSession
= queueConnection.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
```

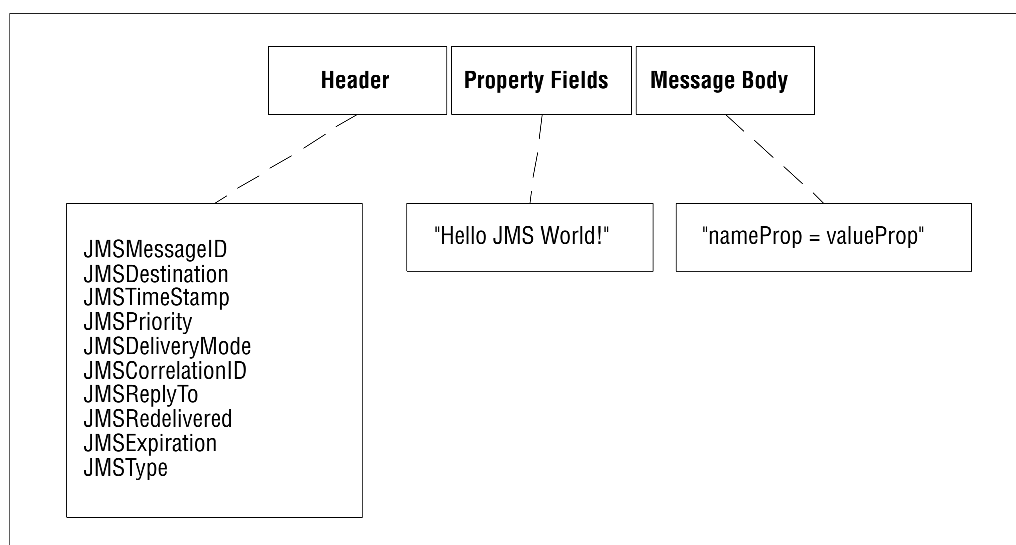
Nel caso di publish/subscribe, la Session viene identificata tramite intefaccia TopicSession:

```
TopicSession topicSession = topicConnection.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);
```

Messages

I messaggi JMS sono costituiti da tre parti: un'intestazione (*message header*), una serie di campi contenenti proprietà (*property fields*) e il corpo (*message body*).

Figura 12.5 – *Messaggio JMS.*



Il message header è costituito esattamente da 10 campi obbligatoriamente presenti. Tali campi sono valorizzati dal provider o dal client al fine di identificare, configurare e inoltrare il messaggio.

Tra questi campi è presente l'identificatore univoco del pacchetto (JMSMessageID), il nome della Queue o Topic a cui il pacchetto è destinato(JMSDestination), la priorità (JMSPriority), il timestamp del tempo d'inoltro(JMSTimestamp), il flag indicante se il messaggio è stato re-inoltrato(JMSRedelivered), l'ID per correlare i messaggi JMS tra di loro (JMSCorrelationID), la modalità di invio(JMSdeliveryMode), la destinazione a cui inoltrare il messaggio di risposta(JMSReplyTo).

I campi JMSReplyTo e JMSCorrelationID e il campo opzionale JMSType sono assegnati dall'applicazione JMS in modo esplicito sull'oggetto Message; tutti gli altri, invece, dal provider JMS.

I property fields sono coppie di nome e valore e sono utili per costruire "filtri" a livello applicativo. La caratteristica di questi campi è che possono essere esaminati mediante i *message selectors*.

Quando un consumatore si connette al server può definire un message selector il quale esamina l'header e i property fields – non il body message – del pacchetto JMS di una specifica destinazione sia essa una Queue o un Topic.

Questo permette alle applicazioni JMS di utilizzare i message selector per specificare quali messaggi è interessata a ricevere utilizzando una sintassi condizionale sottoinsieme del linguaggio SQL-92. Il filtraggio avviene a livello di server e permette di inoltrare ai client lungo la rete i messaggi strettamente necessari o utili, risparmiando così la banda del canale. Un message selector non è altro che un oggetto di classe String che può contenere boolean literals (TRUE, true, FALSE e false), operatori logici (NOT, AND, OR), aritmetici (+, –, *, /), di comparazione(=, >, >=, <, <=, <>), di ricerca (IS, LIKE, IN, NOT IN) e così via [JAPI]. Esempi di message selector:

```
"squadra = 'Pittsburgh Steelers'"
"squadra <> 'Dallas Cowboys'"
"age NOT BETWEEN 20 and 32"
"type = 'FIAT' AND color = 'blue' AND weight >= 100"
"NewsType = 'Opinion' OR NewsType = 'Sports'"
```

I property field sono valorizzabili mediante i metodi setObjectProperty() e getObjectProperty(). Il metodo setObjectProperty accetta in ingresso valori di classe Boolean, Byte, Short, Integer, Long, Float, Double e String e, se si cerca di utilizzare qualsiasi altra classe, viene generata l'eccezione JMSException.

Le proprietà possono essere lette mediante i metodi get<Class>Property; nel caso di proprietà inesistente i metodi getStringProperty e getObjectProperty ritornano null; per le altre classi viene sollevata una NullPointerException. Infine le proprietà possono essere cancellate mediante il metodo clearProperties(), lasciando il messaggio JMS con il campo PropertyFields vuoto.

Body Message

I messaggi JMS possono essere di varie tipi, ognuno dei quali mette a disposizione gli appositi metodi get e set (tab. 12.1). JMS non supporta uno specifico body per messaggi XML; in questi casi è utilizzabile il text message. I messaggi vengono creati mediante i metodi dell'interfaccia Session dalla quale estendono sia l'interfaccia QueueSession che TopicSession.

L'interfaccia mette a disposizione i metodi di creazione relativi a ogni tipologia di message: createTextMessage, createObjectMessage, createMapMessage, createBytesMessage e createStreamMessage,

Tabella 12.1 – *Tipologie di messaggi JMS.*

Tipo di messaggio	Descrizione del payload
TextMessage	Contengono messaggi di tipo String. Sono manipolabili mediante i metodi setText(String s) e getText().
ObjectMessage	Contengono oggetti Java serializzabili. Sono manipolabili mediante i metodi setObject(Object o) e getObject().
MapMessage	Contengono coppie di valori nome/valore. Sono manipolabili mediante i metodi setString(key, value) e getString(key) che restituisce il valore corrispondente alla chiave specificata. I metodi getter e setter sono disponibili per ogni specifica classe di riferimento (setBoolean, getBoolean, setInt, getInt, setObject, getObject, …).
BytesMessage	Contengono array di tipi primitivi.
StreamMessage	Contengono stream di valori di tipo primitivo Java (int, double, char, e così via) e sia la scrittura che la lettura avvengono in modo sequenziale.

prevedendo per ognuno di essi la versione *overloaded* che contempla come parametro in ingresso un oggetto di inizializzazione.

Per creare messaggi, nel caso di tipologia point-to-point è sufficiente invocare il metodo create opportuno sulla QueueSession, mentre nel caso di publish/subscribe sull'oggetto TopicConnection come mostrato negli esempi che seguono:

```
TextMessage txtMsg = queueSession.createTextMessage();
txtMsg.setText("Hello JMS PTP World!");
TextMessage txtMsg2 = queueSession.createTextMessage("Hello JMS PTP World!");
ObjectMessage objMsg = queueSession.createObjectMessage();
objMsg.setObject(my_ptp_order);
ObjectMessage objMsg2 = topicSession.createObjectMessage();
ObjMsg2.setObject(my_pubSub_chat_msg);
```

MessageProducer

Il MessageProducer è l'oggetto che permette l'invio dei messaggi verso una particolare destinazione. Utilizzando gli oggetti Session e Destination creati, si possono inizializzare uno o più message producer.

Nel caso point-to-point, il MessageProducer per inviare messaggi sulla Queue è referenziato mediante l'interfaccia QueueSender:

```
QueueSender queueSender = queueSession.createSender(queue);
```

mentre nel caso di publish/subscribe l'invio dei messaggi verso il Topic può avvenire mediante l'intefaccia TopicPublisher:

```
TopicPublisher topicPublisher= topicSession.createPublisher(topic);
```

Una volta creato il produttore, basta invocare il metodo send sull'oggetto QueueSender nel caso PTP, mentre nel caso pub/sub, il metodo publish va invocato sull'oggetto topicPublisher; entrambi i metodi send richiedono in ingresso un parametro d'interfaccia Message e presentano firme overloaded.

```
queueSender.send(message);
topicPublisher.publish(message);
```

MessageConsumer

Il MessageConsumer rappresenta un oggetto in grado di ricevere messaggi. Analogamente a quanto avviene per il MessageProducer, bisogna indicare l'oggetto Session e la Destination d'interesse per inizializzare correttamente il MessageConsumer.

```
QueueReceiver queueReceiver = queueSession.createReceiver(queue);
TopicSubscriber topicSubscriber = topicSession.createSubscriber(topic);
```

Nella modalità asincrona, il MessageProducer (sender o publisher) non è tenuto ad attendere che il messaggio sia ricevuto per proseguire nel suo funzionamento, e l'elaborazione del messaggio stesso non è necessariamente sequenziale.

Il MessageConsumer è in grado di ricevere i messaggi in modo asincrono definendo un message listener: si tratta di una classe che implementa l'interfaccia javax.jms.MessageListener. Ogni qualvolta un messaggio arriva alla destinazione d'interesse viene automaticamente invocato il metodo di callback onMessage().

Anzitutto si deve creare un listener, cioè un oggetto d'interfaccia javax.jms.MessageListener:

```
TopicListener topicListener = new TextListener();
QueueListener queueListener = new TextListener();
```

e passarlo come argomento al metodi setMessageListener dell'oggetto Receiver

```
topicSubscriber.setMessageListener(topicListener);
queueReceiver.setMessageListener(queueListener);
```

Connesso il listener ci si pone in ascolto, in attesa di ricevere ed elaborare i messaggi, invocando il metodo start sull'oggetto Connection:

```
queueConnection.start();
topicConnection.start();
```

La classe Listener è una normale classe Java che implementa l'interfaccia JMS MessageListener ridefinendo perciò il metodo onMessage(). In tale metodo si mette la logica applicativa per

gestire il contenuto del messaggio ricevuto; si effettua il controllo del tipo del messaggio ricevuto e, se è del tipo corretto, se ne elabora il contenuto.

```
public class TextListener implements MessageListener {
  public void onMessage(Message message) {
    try {
      if (message instanceof TextMessage) {
        TextMessage  txtMsg = (TextMessage) message;
        System.out.println("Ricevuto: " + txtMsg.getText());
        . . .
      }
      else if (message instanceof ObjectMessage) {
        objMsg = (ObjectMessage) message;
        MyMsgClass msg = (MyMsgClass)obj.getObject();
        // MyMsgClass è una classe proprietaria serializzabile
        . . .
```

I prodotti di messaging sono intrinsecamente asincroni ma è bene specificare che un MessageConsumer può ricevere i messaggi anche in modo sincrono. Nella ricezione dei messaggi in modo sincrono il consumatore richiede esplicitamente alla destinazione di prelevare il messaggio (*fetch*) invocando il metodo receive.

Il metodo receive() appartiene all'interaccia javax.jms.MessageConsumer (dalla quale estendono sia QueueReceiver che TopicReceiver) ed è sospensivo, cioè rimane bloccato fino alla ricezione del messaggio, a meno che non si espliciti un timeout finito il quale il metodo termina:

```
public Message receive() throws JMSException
public Message receive(long timeout) throws JMSException
```

Esempio di ricezione sincrona:

```
while(<condition>) {
  Message m = queueReceiver.receive(1000);
  if( (m!=null)&&(m instanceof TextMessage) {
    message = (TextMessage) m;
      System.out.println("Rx:" + message.getText());
  }
}
```

Finite le operazioni di gestione del messaggio ricevuto, si esegue il cosiddetto codice di "pulizia" chiudendo la connessione JMS mediante il metodo close.

```
queueConnection.close();
topicConnection.close();
```

Gestione degli administered objects

Gli *administered objects* (AO) sono risorse *vendor-dependent*: cioè la loro implementazione varia da provider a provider, ed è per questo motivo che non sono gestite a livello di codice. Il programmatore Java fa riferimento ai nomi logici JNDI associati alle risorse astraendosi così dai dettagli implementativi dipendenti dallo specifico application server.

Andando nel concreto, nel codice Java di applicazioni JMS si ricercano (*lookup*) le risorse "fisiche" a cui sono associati nomi logici. Ad esempio, il seguente frammento di codice cerca di recuparare un AO di classe TopicConnectionFactory e una destinazione Topic aventi rispettivamente i nomi logici MokaTopic e MokaTopicConnectionFactory.

```
Context jndiContext = new InitialContext();
TopicConnectionFactory topicConnectionFactory = (TopicConnectionFactory)jndiContext.lookup("MokaTopciConnectionFactory");
Topic topic = (Topic)jndiContext.lookup("topic/MokaTopic");
```

Occorre quindi creare sul provider JMS gli AO opportuni, cioè le risorse "fisiche" da associare ai nomi logici MokaConnectionFactory e MokaTopic.

Nel caso della Sun Reference Implementation (RI), la creazione degli AO si traduce nell'utilizzo del tool j2eeadmin (<J2EE_HOME>/bin) eseguibile da console, specificando i flag addJmsFactory per creare i ConnectionFactory e addJmsDestination per creare le Destination.

```
j2eeadmin -addJmsFactory MokaTopicConnectionFactory topic
j2eeadmin -addJmsDestination topic/MokaTopic topic
```

Altri JMS Provider mettono a disposizione delle console d'amministrazione grafiche o accessibili mediante web come nel caso di Weblogic. Nel caso di JBoss – l'application server JBoss include il JMS Provider JBossMQ – bisogna invece intervenire sul file XML di configurazione.

```
<mbean code="org.jboss.mq.server.jmx.Topic"
 name="jboss.mq.destination:service=Topic,name=MokaTopic">
  <depends optional-attribute-name="DestinationManager">jboss.mq:service=DestinationManager</depends>
  <depends optional-attribute-name="SecurityManager">jboss.mq:service=SecurityManager</depends>
</mbean>
```

Dopo avere capito che ogni produttore ha un modo proprietario di creare e amministrare i propri AO, è importante parlare di come procedere alla loro individuazione.

Lookup degli administered objects

Ogni prodotto usa un proprio specifico protocollo per reperire e utilizzare gli AO. Mediante le API JNDI è possibile utilizzare diverse tipologie di protocolli a parità di codice Java. Da programma basta invocare il metodo lookup sull'oggetto jndiContext di classe javax.naming.Context, specificando il nome logico della risorsa da ricercare, ed effettuare l'opportuno downcast.

```
TopicConnectionFactory topicConnectionFactory
= (TopicConnectionFactory)jndiContext.lookup("MokaTopciConnectionFactory");
Topic topic = (Topic)jndiContext.lookup("topic/MokaTopic");
```

Per reperire opportunamente gli AO bisogna specificare l'URL del Service Provider (il JMS provider) e la classe che fornisce l'implementazione del protocollo.

Queste informazioni sono definite come proprietà di sistema e più precisamente, per quanto riguarda il protocollo, bisogna configurare la proprietà di sistema java.naming.factory.initial (javax.naming.Context.INITIAL_CONTEXT_FACTORY) mentre, per il provider URL, la proprietà java.naming.provider.url (javax.naming.Context.PROVIDER_URL). Ad esempio nel caso della Sun RI l'Initial Context Factory deve essere valorizzato a com.sun .enterprise.naming.SerialInitContextFactory mentre, nel caso di JBoss, a org.jnp.interfaces .NamingContextFactory.

Il Context Provider URL invece indica l'indirizzo dove è disponibile il JMS provider.
Ad esempio nel caso della Sun RI è del tipo iiop://<SERVER>:1050 mentre nel caso di JBoss è jnp:/ /<<SERVER>>:1099, dove al posto di <SERVER> occorre specificare il nome o l'indirizzo IP del JMS Server (ad esempio localhost o 127.0.0.1 nel caso in cui l'application server sia installato sulla macchina locale). La valorizzazione di queste due proprietà può avvenire da codice, da riga di comando oppure da file di property.

Da codice è possibile referenziare le proprietà mediante le proprietà pubbliche e statiche della classe javax.naming.Context, Context.INITIAL_CONTEXT_FACTORY e Context.PROVIDER_URL. Ad esempio, nel caso di Weblogic, il seguente codice permette di utilizzare il protocollo proprietario BEA denominato T3, la cui implementazione è fornita dalla classe di nome weblogic.jndi .WLInitialContextFactory interrogando il server di nome manowar in ascolto sulla porta TCP 7001:

```
java.util.Properties properties = new java.util.Properties();
properties.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WLInitialContextFactory");
properties.put(Context.PROVIDER_URL, "t3://manowar:7001");
Context jndiContext = new InitialContext(properties);
```

Piuttosto che valorizzare il Context Factory e il Provider URL da codice, si può effettuare la configurazione mediante il flag -D da riga di comando in fase di lancio dell'interprete java:

```
-Djava.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
-Djava.naming.provider.url=t3://<SERVER>:7001
```

In questo caso, nel codice si istanzia il Context mediante il costruttore di default della classe InitialContext:

```
Context jndiContext = new InitialContext();
```

Oltre a valorizzare le proprietà JNDI bisogna indicare nel classpath i JAR contenenti le classi che forniscono sia l'implementazione del Context Factory che del protocollo di comunicazione.

Sempre nel caso di Weblogic, nel classpath bisogna indicare il weblogic.jar che contiene sia le classi JMS sia quelle del driver di comunicazione che implementa il protocollo T3:

```
%JAVA_HOME%\bin\java -cp .;classes;%BEA_HOME%\wlserver6.1\lib\weblogic.jar
-Djava.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
-Djava.naming.provider.url=t3://<SERVER>:7001
it.mokabyte.jms.chat.ChatGui
```

Tale comando va specificato su un'unica riga e al posto di <SERVER> va specificato o il nome o l'indirizzo IP del Server JMS.

Una semplice applicazione PTP

Come già detto, un'applicazione JMS è caratterizzata da messaggi (oggetti di classe TextMessage, ObjectMessage, BytesMessage, MapMessage, StreamMessage), produttori di messaggi (oggetti di classe QueueSender o TopicPublisher) e consumatori di messaggi (oggetti di classe QueueReceiver o TopicSubscriber).

JMS espone un modello di programmazione piuttosto semplice, basato sull'esecuzione predeterminata di passi: localizzazione del ConnectionFactory e della destinazione, creazione di una connessione e di una sessione JMS, creazione del produttore e/o consumatore, creazione del messaggio da inviare, invio e/o ricezione del messaggio.

Iniziamo quindi a sviluppare una semplice applicazione JMS che utilizza il modello point-to-point e la modalità di ricezione sincrona.

Scopo dell'applicazione è permettere lo scambio di messaggi TextMessage tra un QueueSender (la classe HelloPtpSender) e un QueueReceiver (la classe HelloPtpReceiver).

Il produttore PTP: la classe HelloPtptSender

La classe HelloPtpSender ha lo scopo di inviare *n* messaggi di tipo TextMessage a una destinazione ovviamento di tipo queue.

La prima operazione che viene effettuate nel costruttore è creare un JNDI Context per reperire gli AO mediante le opportune lookup.

```
Context jndiContext = new InitialContext();
this.queueConnectionFactory
= (QueueConnectionFactory)jndiContext.lookup("MokaQueueConnectionFactory");
this.queue = (Queue) jndiContext.lookup("MokaQueue");
```

Viene quindi creata una connessione con il JMS Server invocando sull'oggetto factory il metodo createQueueConnection().

```
queueConnection = this.queueConnectionFactory.createQueueConnection();
```

Dall'oggetto queueConnection viene creata una sessione mediante il metodo createQueueSession()

```
QueueSession queueSession = queueConnection.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
```

A questo punto si costruisce il produttore di messaggi con il metodo QueueSession.createSender() indicando come parametro la destinazione sulla quale indirizzare i messaggi prodotti

```
QueueSender queueSender = queueSession.createSender(queue);
```

Si procede poi creando il messaggio di classe TextMessage

```
TextMessage message = queueSession.createTextMessage();
```

per inviarlo un numero di volte prestabilito alla coda

```
for(int i=0; i<MAX; ++i) {
  // attesa di un tempo casuale
  try {
    Thread.sleep((int)(Math.random() * Defines.MAX_TIME_TO_SLEEP ));
  } catch (InterruptedException ie) {System.err.println(ie.getMessage());}
  // inizializzazione del messaggio
  message.setText("#" + i + "# Hello JMS Point to Point World: " + new java.util.Date().toString());
  // invio del messaggio
  queueSender.send(message);
}
```

Il consumatore PTP: la classe HelloPtptReceiver

Il consumatore, una volta individuata la coda, crea la conessione e la sessione JMS in modo analogo a quanto visto per l'HelloPtpSender, per poi creare un ricevitore di classe QueueReceiver

```
queueConnection = this.queueConnectionFactory.createQueueConnection();
QueueSession queueSession = queueConnection.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
QueueReceiver queueReceiver = queueSession.createReceiver(queue);
queueConnection.start();
```

che si pone in attesa di ricevere i messaggi sulla coda in modalità sincrona con un timeout d'attesa di 5 secondi.

```
while (true) {
Message m = queueReceiver.receive(5000);
```

Se il messaggio ricevuto è valido e di classe TextMessage, viene visualizzato a console:

```
if (m != null) {
   if (m instanceof TextMessage) {
      TextMessage message = (TextMessage) m;
      System.out.println("HelloPtpReceiver.doTest: messaggio ricevuto[" + message.getText() + "]");
   }
}
```

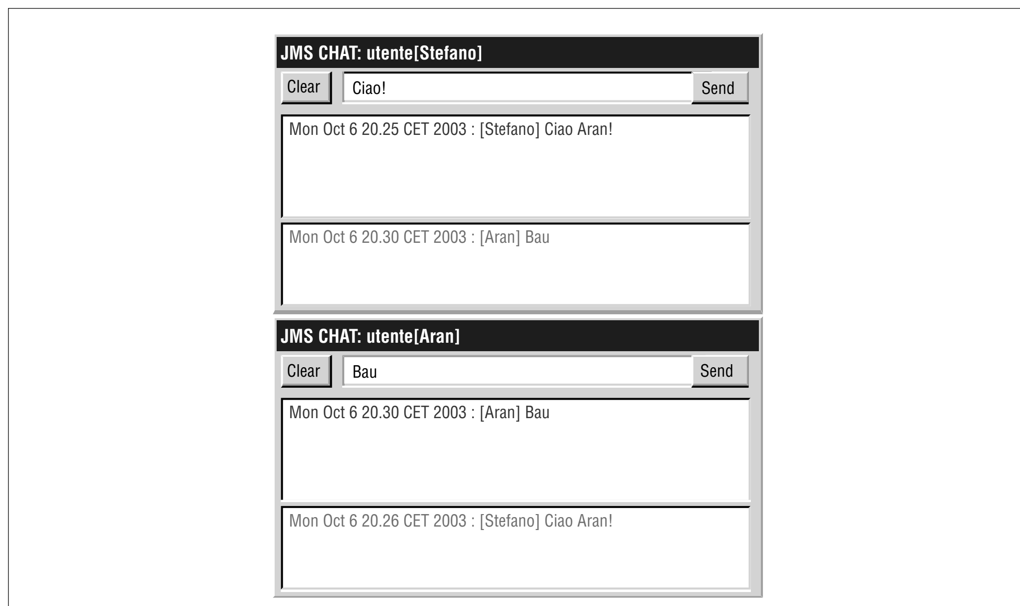
Se ci sono più ricevitori in ascolto sulla stessa coda è possibile che il JMS provider applichi un algoritmo di load balancing. Tale comportamento è dipendente dallo specifico JMS provider.

Un'applicazione publish/subscribe: la chat JMS

Questo esempio propone un'applicazione JMS che utilizza il modello publish/subscribe e la modalità di ricezione asincrona e permette lo scambio tra più produttori e consumatori che condividono la medesima destinazione, realizzando di fatto la classica chat (fig. 12.6).

Il codice dell'esempio è costituito da due classi: il client JMS (JmsChat) e la classe listener adibita alla ricezione asincrona dei messaggi (MyListener). A tale scopo ogni client JMS si dichiara sia publisher che subscriber presso il medesimo topic diventando in grado sia di ricevere che di spedire messaggi. La prima operazione che il costruttore effettua è creare un JNDI Context per potere reperire gli administered object (AO) mediante le opportune lookup.

Figura 12.6 – *La chat JMS in azione.*



```
TopicConnectionFactory topicConnectionFactory
= (TopicConnectionFactory)jndiContext.lookup(topicConnectionFactoryName);
Topic topic = (Topic)jndiContext.lookup(topicName);
```

Nel caso in cui si esegua il programma senza parametri da riga di comando, si ricerca un ConnectionFactory e un Topic rispettivamente di nome MokaConnectionFactory e topic/MokaTopic. Naturalmente i nomi degli AO sono valorizzabili mediante i parametri passati da riga di comando.

```
if(args.length == 0) {
  new JmsChatter("MokaTopicConnectionFactory", "topic/MokaTopic");
}
else if ( (args.length == 1)) {
  new JmsChatter(args[0], "TOPIC/MokaTopic");
}
else if ( (args.length == 2)) {
  new JmsChatter(args[0], args[1]);
}
```

Costruiti e localizzati gli AO, passiamo ora ad analizzare il codice dell'applicazione indipendente dalla tipologia di JMS server.

Mediante l'invocazione del metodo buildGui() si procede a costruire l'interfaccia grafica dell'applicazione che è semplicemente costituita da due bottoni, un text field e due text area. Il text field ha il compito di permettere l'inserimento del messaggio da trasmettere, il bottone bSend gestisce l'invio del messaggio mentre le text area taTx e taRx visualizzano rispettivamente il testo dei messaggi trasmessi e di quelli ricevuti.

Invocando sull'oggetto topicConnectionFactory il metodo createTopicConnection(), viene creata una connessione con il JMS Server.

```
this.topicConnection = topicConnectionFactory.createTopicConnection();
```

Dall'oggetto topicConnection viene creata una sessione mediante il metodo createTopicSession che ha, come parametri in ingresso, un boolean, per specificare se la sessione deve essere gestita in un contesto transazionale, e un intero per indicare la modalità di conferma di ricezione del messaggio:

```
TopicSession topicSession = topicConnection.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);
```

Ci sono tre diverse possibilità di gestione della conferma della ricezione del messaggio mediante il valore del secondo parametro.

- AUTO_ACKNOWLEDGE: la segnalazione dell'acknowledge è automatica e a cura dalla sessione, e avviene subito dopo l'eleborazione del messaggio da parte del ricevente.

- DUPS_OK_ACKNOWLEDGE: la sessione segnalerà l'avvenuta ricezione in un momento imprecisato successivo all'elaborazione del messaggio da parte del ricevente. Questa opzione è "leggera" in termini di overhead di protocollo, però è utilizzabile solo laddove il ricevente possa tollerare la ricezione di messaggi duplicati.

- CLIENT_ACKNOWLEDGE: il client è responsabile della conferma della ricezione dei messaggi mediante l'uso del metodo acknowledge() sugli oggetti Message.

```
public void onMessage(javax.jms.Message message){

  message.acknowledge();
  …
}
```

Successivamente si hanno le istruzioni che dichiarano l'applicazione interessata a ricevere i messaggi dal topic (sottoscrizione). Per prima cosa si costruisce l'oggetto listener di classe MyListener

```
MyListener topicListener = new MyListener(taRx);
```

per poi invocare sull'oggetto TopicSession il metodo createSubscriber, passando come parametro la destinazione d'interesse, al fine di ottenere il TopicSubscriber:

```
topicSubscriber = topicSession.createSubscriber(topic);
```

Nel nostro caso siamo interessati a essere sia publisher che subscriber del topic e questo implica che tutti i messaggi che invieremo verranno anche recapitati a noi stessi.

Per evitare questo effetto di "eco" – nel caso di una chat alquanto indesiderato – basta utilizzare il metodo nella versione overload che prevede oltre al topic, una String e un boolean:

```
public TopicSubscriber createSubscriber(Topic topic, String messageSelector,
                                        boolean noLocal) throws JMSException
```

Specificando come terzo parametro (noLocal) il valore true, si inibisce il recapito dei messaggi pubblicati dal client medesimo.

```
topicSubscriber = topicSession.createSubscriber(topic,null,true);
```

Nel caso in cui si volesse creare un filtro per indicare le condizioni che devono essere rispettate affinché il pacchetto venga recapitato, basta specificare come secondo parametro il Message Selector da utilizzare. Una volta creato il topicSubscriber, mediante il metodo setMessageListener si registra il topic listener di classe MyTextListener:

```
topicSubscriber.setMessageListener(topicListener);
```

La classe MyListener implementa l'interfaccia javax.jms.MessageListener ridefinendo il metodo onMessage() che si limita a stampare nella text area taRx il messaggio ricevuto nel caso sia di classe TextMessage.

```
class MyListener implements MessageListener {

   private javax.swing.JTextArea textArea = null;

   public MyListener(javax.swing.JTextArea ta) {
      this.textArea = ta;
   }

   public void onMessage(Message message) {
      TextMessage msg = null;
      try {
         if (message instanceof TextMessage) {
            msg = (TextMessage) message;
            // stampo a video
            System.out.println(new java.util.Date().toString() + "Ricevuto messaggio: "+ msg.getText());
            // stampo nella TextArea
            this.textArea.append(new java.util.Date().toString() + ":" + msg.getText());
         } else {
            System.out.println("Message of wrong type: " + message.getClass().getName());
         }
      } catch (JMSException e) {
         . . .
      }
   }
}
```

Da notare che la classe JmsChat avrebbe potuto gestire essa stessa la ricezione dei messaggi, implementando direttamente l'interfaccia MessageListener e definendo al suo interno il metodo onMessage():

```
public class JmsChat implements MessageListener {
   . . .
   topicSubscriber.setMessageListener(this);
   . . .
      public void onMessage(Message message) {
      . . .
   System.out.println(msg.getText());
   }
}
```

Manca infine da spiegare come avviene la trasmissione dei messaggi. Premendo il bottone bSend, viene invocato il metodo actionPerformed della classe anonima adibita alla gestione dell'evento. Tale metodo valorizza l'oggetto TextMessage mediante il metodo setText e provvede al suo invio mediante il metodo publish invocato sull'oggetto topicPublisher.

```
bSend.addActionListener(new java.awt.event.ActionListener() {
  public void actionPerformed(ActionEvent e) {
    try{
      message.setText("[" + userName + "] " + tfTx.getText());
      topicPublisher.publish(message);
      // echo to TextArea of data transmitted
      taTx.append(new java.util.Date().toString() + ": " + message.getText());
      . . .
```

È bene aprire infine una parentesi sull'API di creazione delle sessioni JMS. Nel caso si specifichi come valore true il primo parametro del metodo TopicConnection.createTopicSession()

```
TopicSession topicSession = topicConnection.createTopicSession(true,Session.AUTO_ACKNOWLEDGE);
```

si richiede la creazione di un contesto transazionale all'interno della sessione JMS.

Le transazioni create sono dette "locali" (*local transaction*) poiché non coinvolgono un transaction manager JTA-compliant. La demarcazione delle transazioni locali in JMS non prevede un esplicito metodo di begin() per marcare l'inizio di una transazione; tutti i messaggi inviati o ricevuti sono automaticamente raggruppati e gestiti in una transazione. La transazione rimane aperta fino a che non viene eseguito il metodo rollback() o commit() sull'oggetto Session (sia l'interfaccia javax.jms.QueueSession che javax.jms.TopicSession estendono dall'interfaccia javax.jms.Session). Il metodo Session.getTransacted() permette di ottenere come risultato un boolean che indica se la sessione corrente è transazionale o meno.

La possibilità di creare sessioni JMS in un contesto transazionale è consentita anche nel modello PTP, specificando true il primo parametro del metodo QueueConnection.createTopicSession:

```
QueueSession queueSession = queueConnection.createQueueSession(true,Session.AUTO_ACKNOWLEDGE);
```

Nel caso si debbano gestire transazioni distribuite (*two-phase commit*, 2PC), sono disponibili le seguenti API XA: XAConnectionFactory, XAQueueConnectionFactory, XATopicConnectionFactory, XASession, XAQueueSession, XATopicSession. Tali API sono utilizzabili in modo analogo alle corrispondenti versioni non XA fino a ora descritte.

La chat JMS sportiva: i *message selectors*

Partendo dall'esempio precedente di chat, si può pensare di estenderne il funzionamento introducendo la possibilità di gestire gruppi di utenza omogenei. Nel caso specifico, ogni gruppo sarà rappresentato da utenti tifosi della medesima squadra di football americano.

Da un punto di vista JMS, questo scenario si traduce nella propagazione di messaggi dai produttori solo verso i consumatori interessati al messaggio in questione, o più brevemente nello scambio "filtrato" di messaggi tra i partecipanti alla chat.

È necessario quindi creare dei filtri applicativi in grado di "scremare" i messaggi e permetterne l'inoltro in modo appropriato: a livello applicativo, un filtro deve essere specificato sia in fase di creazione del subscriber che in fase di invio del messaggio.

Anche un consumatore quando si connette al server può definire un filtro per poter leggere solo i messaggi ai quali è interessato. Tale filtro, detto *message selector*, è un oggetto di classe String che può contenere valori boolean literals (TRUE, true, FALSE e false), operatori logici(NOT, AND, OR), aritmetici(+, –, *, /), di comparazione (=, >, >=, <, <=, <>), di ricerca (IS, LIKE, IN, NOT IN) e così via, dando vita a una sintassi condizionale sottoinsieme del linguaggio SQL-92.

Il controllo e quindi la selezione avvengono esaminando, mediante i message selector, l'header e i property field – il cosiddetto payload – e non il corpo del messaggio JMS. È importante notare che la selezione avviene sul JMS provider, in modo da inoltrare ai client lungo la rete solamente i messaggi strettamente necessari e/o utili, risparmiando così la banda del canale. I property field sono valorizzabili mediante i metodi setObjectProperty e getObjectProperty. Tornando all'esempio della chat sportiva, si può pensare di definire un message selector che selezioni i messaggi in funzione della squadra a cui fa riferimento il messaggio stesso; ad esempio

```
private String teamName = null;

// Set the Message Selector
String strMessageSelector = "TEAM = '" + this.teamName + "'";
```

La squadra preferita verrà richiesta all'utente in fase di startup dell'applicazione e memorizzata nella proprietà teamName della classe. In fase di creazione del subscriber si deve provvedere a specificare il message selector come secondo parametro del metodo createSubscriber()

```
topicSubscriber =  topicSession.createSubscriber(topic, strMessageSelector, true)
```

mentre prima dell'invio del messaggio si dovrà impostare il valore del property field con il nome della squadra favorita tramite il metodo setStringProperty().

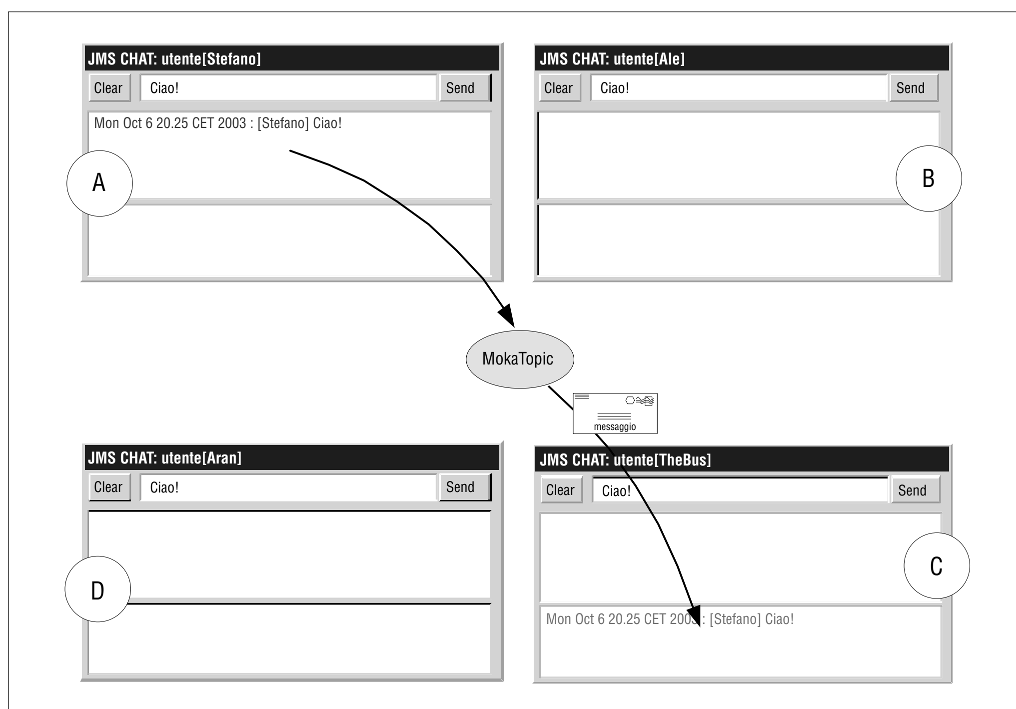
```
// Set the Message Selector in TX
message.setStringProperty("TEAM",this.teamName);
topicPublisher.publish(message);
```

È il ProviderJMS che gestisce direttamente il recapito appropriato dei messaggi (fig. 12.7).

Persistenza dei messaggi

Le API JMS permettono di specificare due possibili modi di inviare i messaggi (persistente e non) mediante i quali si decide se i messaggi verranno perduti in seguito a un crash del JMS provider.

Figura 12.7 – *I messaggi inviati dall'utente A vengono recapitati al solo utente C, essendo i due utenti tifosi della medesima squadra, ma non agli utenti B e D, tifosi di squadre avversarie.*



Queste modalità di invio sono definite tramite i campi dell'interfaccia DeliveryMode. Il delivery mode di default è quello persistente.

Un messaggio inviato con il DeliveryMode.PERSISTENT sarà recapitato una e una sola volta (*once-and-only-once*); ciò assicura che il messaggio non sia perso in caso di errore del JMS provider. È compito del JMS provider stesso memorizzare opportunamente il messaggio e recapitarlo al suo riavvio. Un messaggio inviato con il DeliveryMode.NON_PERSISTENT non viene reso persistente e quindi verrà perso nel caso si verifichi un errore da parte del JMS provider stesso (*at-most-once*).

In fase di stesura del codice è possibile specificare il delivery mode in due modi: il primo si basa sull'utilizzo del metodo setDeliveryMode() dell'interfaccia MessageProducer (da cui estendono sia l'interfaccia QueueSender che TopicPublisher), per configurare la modalità di consegna dei messaggi prodotti da quel produttore.

```
topicPublisher.setDeliveryMode(DeliveryMode.PERSISTENT)
```

Alternativamente, si può impostare la modalità delivery sul singolo messaggio mediante parametro nei metodi send() o publish()

```
topicPublisher.publish(<< message >>, DeliveryMode.NON_PERSISTENT,  <<priority level>>, << expiration time>>);
```

Nel metodo publish(), il terzo e quarto argomento permettono rispettivamente di gestire il livello di priorità del messaggio (priority level) e il suo tempo di vita (expiration time).

Livello di priorità dei messaggi

Se non si specifica la priorità del messaggio in fase di trasmissione, il livello di priorità di default è 4. Per specificare la priorità del messaggio sono possibili due modi.

Il primo consiste nell'invocare il metodo setTimeToLive sull'interfaccia MessageProducer, per specificare il timeToLive valido per tutti i messaggi prodotti da quel produttore.

```
topicPublisher.setPriority(5);
topicPublisher.setPriority(Message.DEFAULT_PRIORITY+1);
```

È possibile altrimenti indicare la priorità per uno specifico messaggio come terzo argomento del metodo send o publish.

```
topicPublisher.publish(<<message>>,<<DeliveryMode>>,3,<<timeToLive>>);
```

Expiration time

Per permettere ai messaggi di "spirare" è possibile specificare la proprietà di *expiration time*. Di default un messaggio ha tempo di vita infinito. Da un punto di vista applicativo, però, si può avere necessità che un messaggio, diventato obsoleto, venga rimosso dal provider.

Per specificare il tempo di vita di un messaggio si può invocare il metodo setPriority() sull'interfaccia MessageProducer, per specificare la priorità valida per tutti i messaggi prodotti da quel produttore, oppure specificare il valore (espresso in millisecondi) come quarto argomento del metodo send() o publish() per lo specifico messaggio.

Specificando un tempo di vita pari a zero, i messaggi non verranno mai marcati come obsoleti. Qualsiasi messaggio che non sia stato inviato entro l'expiration date sarà distrutto.

In definitiva il metodo publish() della classe TopicPublisher permette di specificare i vari parametri di esecuzione grazie alle tre firme:

```
public void publish(Message message)throws JMSException
public void publish(Topic topic, Message message)throws JMSException
public void publish(Message message, int deliveryMode, int priority, long timeToLive) throws JMSException
```

Nel caso PTP, il metodo send() della classe QueueSender mette a disposizione le analoghe firme viste per il metodo TopicPublisher.publish():

```
public void send(Message message) throws JMSException
public void send(Message message, int deliveryMode, int priority, long timeToLive) throws JMSException
```

```
public void send(Queue queue, Message message) throws JMSException
public void send(Queue queue, Message message, int deliveryMode, int priority,
                 long timeToLive) throws JMSException
```

I durable subscribers

Un'importante caratteristica offerta dai sistemi MOM è il concetto di Guaranteed Message Delivery (GMD) che garantisce la consegna del messaggio, anche in caso di problemi al sistema o, peggio ancora, di crash, che porterebbe a una indesiderata perdita di informazioni: è per questo motivo che i messaggi possono essere resi persistenti (per esempio: mediante file o mediante JDBC) prima di essere recapitati ai consumatori. Questo meccanismo prende il nome di *store-and-forward*. Il meccanismo di store deve essere impostato da client (DeliveryMode a PERSISTENT) mentre il compito di store e del relativo forward del messaggio è del JMS provider.

Un TopicSubscriber (al contrario di un QueueReceiver) "ordinario" (cioè non-durable) è però in grado di ricevere i messaggi solo se attivo. È possibile però creare delle "sottoscrizioni persistenti" dette *durable subscriptions* le quali sono in grado di recevere i messaggi anche se il subscriber non era in ascolto al momento della generazione del messaggio.

Questo è possibile creando il subscriber mediante il metodo TopicSession.createDurableSubscriber() e non con il metodo TopicSession.createSubscriber() usato fino a ora che permette di creare solo subscriber non persistenti (*non durable subscriber*).

Nel caso di un subscriber non persistente, se la sottoscrizione avviene al tempo t=0 e finisce al tempo t=3, e successivamente viene creata nuovamente al tempo t=5 e termina al tempo t=6, il messaggio M2 arrivato al tempo t=4 non viene ricevuto dall'applicazione.

Nel caso di durable subscription, il messaggio M2 viene memorizzato dal provider e recapitato al tempo t=5 all'applicazione senza essere perso. La sottoscrizione di tipo persistente deve essere chiusa in modo esplicito invocando il metodo unsubscribe() sull'oggetto TopicSession specificando l'ID della sottoscrizione da rimuovere. La sottoscrizione persistente dura da quando viene creata

```
topicSession.createDurableSubscriber(…);
```

fino a quando viene chiusa

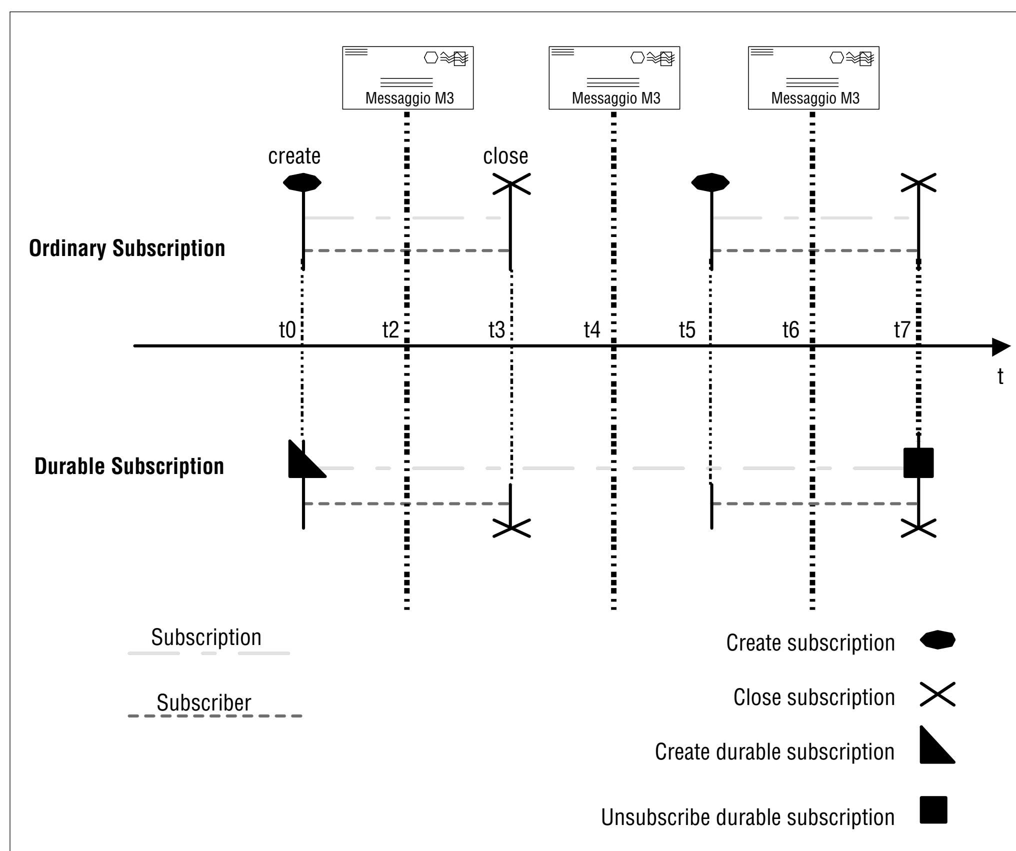
```
topicSession.unsubscribe(…);
```

Dalla fig. 12.8 è possibile vedere la differente durata della vita dei subscriber.

Nel caso in cui le applicazioni JMS non possono tollerare perdite di dati, l'utilizzo delle destinazioni persistenti diventa praticamente d'obbligo. Si deve comunque tenere ben presente che la gestione delle durable subscriptions introduce un overhead, dato che il JMS provider deve provvedere a salvare i messaggi in un qualche sistema di persistenza come un file o un database.

Ad esempio, in JBoss il meccanismo di default di persistenza MQ Server è costituito da un file che permette di memorizzare 1000 messaggi prima di iniziare a scartare i più vecchi (tecnicamente tale gestione è detta persistenza con Rollover File).

Figura 12.8 – *La diversa durata di una sottoscrizione non-durable e durable.*



Creando una durable subscription, specificando come username stefano e identificativo univoco della sottoscrizione 123, viene creato da JbossMQ il file <JBOSS_HOME>/db/jbossmq/TOPIC.MokaDurableTopic.id123-stefano.dat0 dove vengono memorizzati i messaggi pubblicati. In questo modo, i messaggi verranno recapitati anche ai subscriber che non erano attivi al momento dell'invio del messaggio, ma che si collegheranno solo successivamente.

Nel caso in cui si voglia utilizzare un sistema DBMS accessibile mediante JDBC, si dovrà creare l'opportuna tabella che permetta la memorizzazione dell'ID del messaggio – che funzionerà da chiave della tabella –, il nome della destinazione di provenienza e il messaggio da memorizzare; lo script SQL, ad esempio, potrebbe essere qualcosa del tipo

```
CREATE TABLE JMS_MESSAGE
{
  MESSAGEID CHAR(17) NOT NULL,
```

```
  DESTINATION VARCHAR(30) NOT NULL,
  MESSAGEBLOB BLOB,
  PRIMARY KEY (MESSAGEID,DESTINATION)
};
```

Da un punto di vista applicativo, per creare un durable subscriber bisogna invocare sull'oggetto TopicSession il metodo createSubscriber() specificando il topic d'interesse e un nome che identifichi univocamente la sottoscrizione (cioè la durable subscription).

```
public TopicSubscriber createDurableSubscriber(Topic topic, String name)throws JMSException
```

L'id univoco del durable subscriber viene determinato da

- un client id associato alla connessione;
- un topic e un nome che rappresenta l'identificativo univoco della subscriber.

L'identificativo univoco associato al durable subscriber serve al JMS server per memorizzare i messaggi arrivati mentre il subscriber non è attivo. Quando il subscriber si riconnette (logicamente con il medesimo identificativo), il JMS server provvede a inviare tutti i messaggi ancora validi (*unexpired*) accumulati fino a quel momento.

La chat JMS "persistente"

Riconsiderando l'esempio della chat, per implementare la versione durable è necessario innazitutto identificare univocamente una connessione JMS. Questa operazione può variare a seconda del tipo di JMS provider che si utilizza.

Infatti alcuni JMS provider possono richiedere una configurazione statica di tale id o permettere la configurazione dinamica mediante il metodo

```
public void setClientID(java.lang.String clientID) throws JMSException
```

il quale permette di specificare l'identificativo del client da associare alla connessione JMS. Questa permette di associare al cliente la stato della connessione e dei relativi oggetti.

Il metodo setClient() della classe TopicConnection va utilizzato immediatamente dopo aver creato la connessione e prima di qualsiasi operazione che coinvolga lo stesso oggetto Connection. Se utilizzato altrove, verrà sollevata l'eccezione IllegalStateException così come se si cerca di utilizzare un client id già in uso viene lanciata una InvalidClientIDException.

Se si utilizza ad esempio il server WebLogic, è possibile utilizzare il metodo setClientID nel seguente modo:

```
this.topicConnection=topicConnectionFactory.createTopicConnection();
System.out.println("ClientID: "+topicConnection.getClientID());
```

```
// imposta il clientId utilizzando lo userName
this.topicConnection.setClientID(this.userName);
System.out.println("ClientID:"+topicConnection.getClientID());
topicSession =    topicConnection.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);
topicSubscriber= topicSession.createDurableSubscriber(topic,this.userName);
```

Se si omette l'invocazione del metodo setClientID(), si ottiene un'eccezione che indica che la connessione non ha collegato un identificativo valido:

```
# JMSException: weblogic.jms.common.JMSException: Connection clientID is null
weblogic.jms.common.JMSException: Connection clientID is null
. . .
```

Se si provasse a utilizzare JBossMQ, il JMS provider integrato in JBoss, allora l'invocazione del metodo setClientId() solleverebbe un'eccezione IllegalStateException visto che già un ID è stato associato alla connessione da parte del provider stesso: infatti in JBoss l'assegnazione dell'id univoco alla connessione avviene mediante il file di configurazione jbossmq-state.xml, all'interno del quale si deve specificare username, password e client id per lo specifico topic su cui si vuole creare la durable subscription.

Questo meccanismo di configurazione permette meno flessibilità a runtime ma più controllo nella gestione delle risorse. Il codice funzionante con questa modalità di configurazione è:

```
Topic topic = (Topic) jndiContext.lookup("MokaTopic");
this.topicConnection =topicConnectionFactory.createTopicConnection("stefano","mokabyte");
System.out.println("ClientID is:"+topicConnection.getClientID());
topicSession=    topicConnection.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);
topicSubscriber=topicSession.createDurableSubscriber(topic,this.userName);
```

L'applicazione MokaJmsSpy

L'applicazione MokaJmsSpy è un client "spia" JMS che permette di visualizzare alcune utili informazioni (fig. 12.9).

Mediante il pulsante MESSAGE_SETTINGS è possibile ricavarsi i valori di default del messaggio JMS mediante le proprietà static e final della classe javax.jms.Message, e della modalità di invio mediante le proprietà classe javax.jms.DeliveryMode.

```
buttonMsgSet.addActionListener(new java.awt.event.ActionListener() {
  public void actionPerformed(ActionEvent e) {
    String info="Message.DEFAULT_PRIORITY:"+Message.DEFAULT_PRIORITY+"\n";
    info+="Message.DEFAULT_TIME_TO_LIVE:"+Message.DEFAULT_TIME_TO_LIVE+"\n";
    info+="Message.DEFAULT_DELIVERY_MODE:"+Message.DEFAULT_DELIVERY_MODE+"\n";
```

```
        info+="DeliveryMode.NON_PERSISTENT:"+DeliveryMode.NON_PERSISTENT+"\n";
        info+="DeliveryMode.PERSISTENT:"+DeliveryMode.PERSISTENT+"\n";
        JOptionPane.showMessageDialog(null, info, "Message settings", JOptionPane.INFORMATION_MESSAGE);
    }
});
```

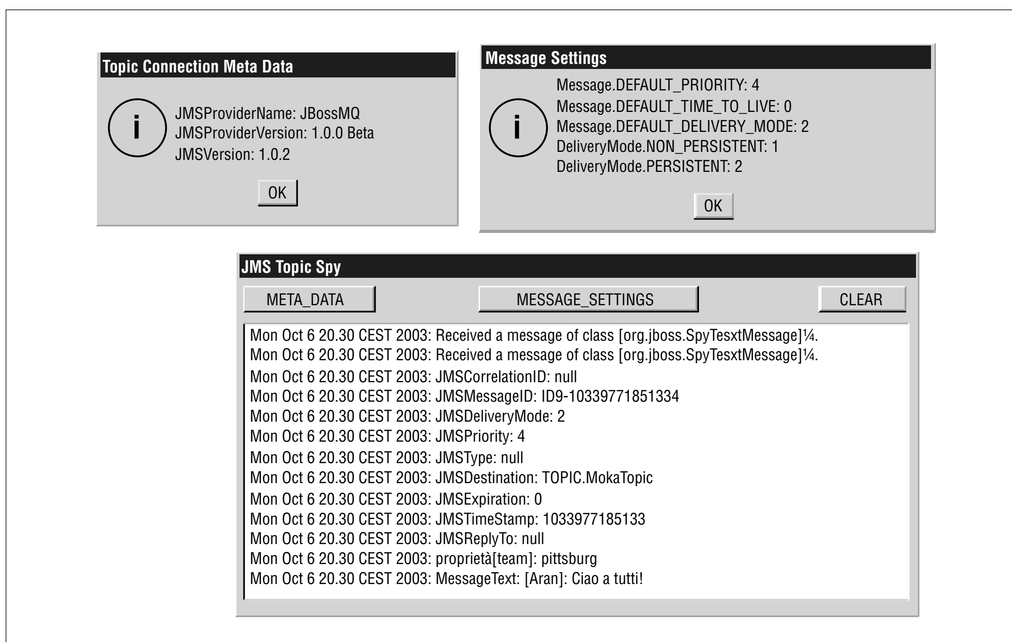
Il pulsante MESSAGE_DATA permette di ricavare i dati descrittivi della connessione JMS; in questo caso si ottengono i metadata della connessione (javax.jms.ConnectionMetaData) invocando il metodo getMetaData() sull'oggetto TopicConnection.

```
buttonMetaData.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(ActionEvent e) {
        try{
            ConnectionMetaData connMetaData = topicConnection.getMetaData();
            String info = "JMSProviderName: " + connMetaData.getJMSProviderName()+"\n";
            info += "JMSProviderVersion: " + connMetaData.getProviderVersion() + "\n";
            info += "JMSVersion: " + connMetaData.getJMSVersion() + "\n";
            JOptionPane.showMessageDialog(null, info, "Topic Connection Meta Data",
                                        JOptionPane.INFORMATION_MESSAGE);
        } catch(JMSException jmse) {. . .}
    }
});
```

I valori delle proprietà e del body del messaggio ricevuto possono essere ottenuti mediante i metodi dell'oggetto Message:

```
this.append("Received message class["+ message.getClass().getName()+"]");
this.append("JMSCorrelationID: " + message.getJMSCorrelationID());
this.append("JMSMessageID: " + message.getJMSMessageID());
this.append("JMSDeliveryMode: " + message.getJMSDeliveryMode());
this.append("JMSPriority: " + message.getJMSPriority());
this.append("JMSType: " + message.getJMSType());
this.append("JMSDestination: " + message.getJMSDestination());
this.append("JMSExpiration: " + message.getJMSExpiration());
this.append("JMSTimestamp: " + message.getJMSTimestamp());
this.append("JMSReplyTo: " + message.getJMSReplyTo());
this.append("JMSRedelivered: " + message.getJMSRedelivered());
java.util.Enumeration enum = message.getPropertyNames();
while(enum.hasMoreElements()){
    String str = (String)enum.nextElement();
    this.append("proprietà["+str+"]: " + message.getObjectProperty(str));
}
```

Figura 12.9 – *L'applicazione MokaJmsSpy in azione.*



La lista dei messaggi presenti nella coda, su cui l'applicazione si è registrata come listener, può essere ricavata mediante la classe QueueBrowser. Tale classe molto utile (disponibile per il solo modello PTP) permette di analizzare i messaggi non ancora consumati presenti in una certa coda senza rimuoverli.

```
QueueBrowser.addActionListener(new java.awt.event.ActionListener() {
  public void actionPerformed(ActionEvent e) {
    try{
      textArea.setText("");
      QueueSession queueSession =
      queueConnection.createQueueSession(
      false,Session.AUTO_ACKNOWLEDGE);
      QueueBrowser browser = queueSession.createBrowser(queue);
      java.util.Enumeration enum = browser.getEnumeration();
      int cnt = 0;
      while(enum.hasMoreElements()){
        append("QueueBrowser ### Messaggio numero "+ (++cnt) + " ###");
        Message message = (Message) enum.nextElement();
```

```
            dumpMessageHeaderProperties(message);
            TextMessage txtMessage = (TextMessage) message;
            append("TextMessage: " + txtMessage.getText());
          }
          append("Messages in ["+queue+"] = " + cnt);
          browser.close();
        }
        catch(JMSException jmse) {. . .}
      }
    });
```

Un'applicazione di compravendita biglietti

Per riepilogare e assimilare i concetti visti fino ad ora, si presenta un'applicazione JMS che ha lo scopo di gestire la vendita di biglietti di partite di football americano con le relative applicazioni client (potenziali acquirenti). L'applicazione (fig. 12.10) è costituita da tre classi:

- TicketMessage: è la classe che incapsula il biglietto della partita da vendere/acquistare;
- TicketSeller: è la classe responsabile di vendere i biglietti delle partite;
- TicketBuyer: è la classe in grado di comprare i biglietti delle partite d'interesse;

La classe TicketMessage incapsula la descrizione del biglietto e per ogni sua proprietà mette a disposizione i relativi metodi get e set:

```
public class TicketMessage implements java.io.Serializable{
  // valore dello stato del biglietto in vendita
  public static final String INSALE = "TICKET IN SALE...";
  // valore dello stato del biglietto comprato
  public static final String SOLD = "! SOLD !";
  /** ID del biglietto */
  private int id;
  /** Stadio */
  private String stadium;
  /** Nome della squadra che gioca in casa */
  private String home;
  /** Nome della squadra ospite */
  private String visitors;
  /** Data della partita */
  private String date;
  /** Stato del biglietto: INSALE / SOLD */
```

```
    private String status;
    /** Data di messa in vendita */
    private String insaleDate;
    /** Data di effettiva vendita del biglietto */
    private String soldDate;
    /** ID del compratore */
    private String userId;
     . . .
```

La vendita dei biglietti

La situazione di vendita è un tipico caso di modellazione publish/subscribe: il venditore, rappresentato dalla classe TicketSeller, pubblica presso un topic(MokaTopicSell) i biglietti da vendere. Chiunque si dichiari subscriber di quel topic è in grado di ricevere gli annunci di vendita. I biglietti da vendere sono contenuti in un array di oggetti di classe TicketMessage:

```
private static final TicketMessage arrayTickets[] = {
    new TicketMessage(0,"Soldier Field","Pittsburgh","Chicago","01/10/02"),
    new TicketMessage(1,"Coliseum","Oakland","Chicago","03/11/02"),
    new TicketMessage(2,"Heinz Field","Pittsburgh","Cincinnati","10/11/02"),
    . . .
}
```

La classe TicketSeller provvede a metterli in vendita, pubblicandoli a istanti temporali definiti se non sono stati nel frattempo già venduti, presso il MokaTopicSell:

```
public class TicketSeller extends JFrame implements MessageListener {
```

Figura 12.10 – *L'applicazione JMS che vende i biglietti e i relativi client acquirenti.*

Ticket Seller

Sell date	Item to sell	Status	User	Date bought
Mon Oct 6 20.30 CET 2003	Washington vs Dallas	Ticket on sale		
x	Oakland vs Chicago	Sold	Aran	Mon Oct 6 18.30 CET 2003
x	Pittsburg vs Cincinnati	Sold	Ste	Mon Oct 6 21.30 CET 2003
Mon Oct 6 21.30 CET 2003	Seattle vs San Francisco	Ticket on sale		
x	Pittsburg vs Cleveland	Sold	Ste	Mon Oct 6 21.30 CET 2003

Ticket Buyer: Aran

Item received	Action performed	Where to buy
Washington vs Dallas	Bad deal don't buy!	TOPIC.JMS_TT7
Oakland vs Chicago	I WANNA BUY!	TOPIC.JMS_TT7
Pittsburg vs Cincinnati	Bad deal don't buy!	TOPIC.JMS_TT7
Seattle vs San Francisco	Bad deal don't buy!	TOPIC.JMS_TT7
Pittsburg vs Cleveland	Bad deal don't buy!	TOPIC.JMS_TT7

Ticket Buyer: Ste

Item received	Action performed	Where to buy
Washington vs Dallas	Bad deal don't buy!	TOPIC.JMS_TT7
Oakland vs Chicago	Bad deal don't buy!	TOPIC.JMS_TT7
Pittsburg vs Cincinnati	I WANNA BUY!	TOPIC.JMS_TT7
Seattle vs San Francisco	Bad deal don't buy!	TOPIC.JMS_TT7
Pittsburg vs Cleveland	I WANNA BUY!	TOPIC.JMS_TT7

```
private void publishTicketsToSell(TopicSession topicSession) {
  this.message = topicSession.createObjectMessage();
   . . .
  for(int i=0; i<arrayTickets.length; ++i) {
    if(arrayTickets[i].getStatus().equals(TicketMessage.INSALE)) { // se il biglietto è in vendita
    try{
      Thread.sleep(5000); // aspetto…
    } catch(InterruptedException ie){. . .}
    // preparo il messaggio da vendere
    arrayTickets[i].setInsaleDate(new java.util.Date().toString());
    message.setObject(arrayTickets[i]);
    message.setJMSReplyTo(tempOrderTopic);
    topicPublisher.publish(message);  // lo pubblico presso il MokaTopicSell
    . . .
```

I compratori sono le applicazioni JMS che si dichiarano subscriber presso il MokaTopicSell.

```
public class TicketBuyer extends JFrame implements MessageListener {
  private void setup() {
    . . .
    TopicSubscriber topicSubscriber=topicSession.createSubscriber(topic,null,true);
    topicSubscriber.setMessageListener(this);
    . . .
```

Fino a qui, niente di nuovo rispetto a quello già visto nei paragrafi precedenti; passiamo a vedere come implementare la parte relativa all'acquisto dei biglietti.

L'acquisto dei biglietti

Tutti i client JMS in ascolto sul topic MokaTopicSell sono potenziali acquirenti. Quando un client JMS decide di comprare un biglietto, deve comunicare con il solo venditore inoltrando la richiesta d'acquisto.

Utilizzo delle temporary destination

Fino ad adesso si sono utilizzate destinazioni gestite come administered object, cioè create/configurate/rimosse dal provider JMS mediante l'opportuno tool di amministrazione oppure lo specifico file di configurazione.

Le API JMS permettono di creare destinazioni dette temporanee (TemporaryQueue e TemporaryTopic) la cui durata è strettamente legata a quella della connessione tramite la quale sono state create. Queste destinazioni sono create dinamicamente dall'applicazione utilizzando i metodi createTemporaryQueue() della classe QueueSession e createTemporaryTopic() della classe TopicSession.

Le destinazioni temporanee sono a uso privato dell'applicazione JMS che le ha create e quindi gli altri client JMS non ne hanno visibilità. Se si chiude la connessione da cui si è creata la destinazione temporanea, anch'essa viene chiusa e il suo contenuto viene perso. Nell'appli-

cazione in esame il TicketSeller crea quindi un temporary topic sul quale si predisporrà a ricevere le richieste di acquisto:

```
tempOrderTopic = topicSession.createTemporaryTopic();
topicSubscriber = topicSession.createSubscriber(tempOrderTopic);
topicSubscriber.setMessageListener(this);
```

Gli acquirenti quindi potranno inviare le richieste di acquisto al topic temporaneo in modo tale che esse siano visibili al solo venditore e non agli altri acquirenti.

Resta da capire come è possibile indicare al consumatore la destinazione del produttore. Per fare questo ci viene in aiuto il campo JMSReplyTo appartenente all'header del messaggio JMS. Tale campo permette al consumatore di un messaggio di ricavare la destinazione a cui rispondere. Nel nostro esempio, l'acquirente può quindi ricavarsi dalla lettura del campo JMSReplyTo la destinazione del venditore verso il quale inoltrare la richiesta d'acquisto.

Per attuare questo semplice meccanismo di request/reply, il client Seller, prima di inviare il messaggio, valorizza il campo JMSReplyTo con il valore del temporary topic precedentemente creato:

```
message.setJMSReplyTo(tempOrderTopic);
topicPublisher.publish(message);
```

Il ricevente legge il campo JMSReplyTo

```
Destination dest = message.getJMSReplyTo();
```

ed effettua l'opportuno downcast alla specifica tipologia della destinazione

```
Topic buytopic = (Topic)dest;
```

per poi provvedere all'invio del messaggio della richiesta d'acquisto

```
publisher = this.topicSession.createPublisher(buytopic); // creo il publisher
publisher.publish(msg); // invio il messaggio d'acquisto
```

Il campo JMSReplyTo permette di fatto un "routing applicativo" di messaggi che può essere particolarmente utile in tipici contesti di workflow in cui è necessario specificare le destinazioni da utilizzare a seconda dello stato e del punto particolare del processo.

Il metodo decideToBuy() conterrà la logica responsabile dell'acquisto del biglietto e verrà invocato ogni qualvolta si riceve un annuncio di vendita.

```
public void onMessage(javax.jms.Message message){
  decideToBuy(message);
}
```

Tale metodo effettua un semplice confronto tra la squadra che gioca in casa indicata nel biglietto da comprare (ticket.getHome()) e la squadra preferita dell'utente (this.teamName)

```
private void decideToBuy(Message message){
  ObjectMessage msg = null;
  try {
    if (message instanceof ObjectMessage) {
      msg = (ObjectMessage) message;
      TicketMessage ticket = (TicketMessage)msg.getObject();
      if(ticket.getHome().equalsIgnoreCase(this.teamName)){
```

Se i nomi delle squadre sono uguali, si valorizza il corpo del messaggio con i dati del biglietto d'interesse

```
msg.setObject(ticket);
```

e viene ricavata la destinazione alla quale inoltrare la richiesta d'acquisto

```
Topic buytopic =(javax.jms.Topic)message.getJMSReplyTo();
```

per poi inviare il messaggio

```
publisher = this.topicSession.createPublisher(buytopic);
publisher.publish(msg);
. . .
```

Utilizzo delle code in modalità point-to-point

Mentre la modellazione di vendita dei biglietti è un tipico caso di relazione uno-a-molti (*1* venditore – *n* acquirenti), l'acquisto dei biglietti è di fatto una relazione uno-a-uno tra acquirente e venditore e quindi un tipico scenario point-to-point.

Il TicketSeller effettua quindi una lookup sulla coda MokaQueueBuy e si pone in ricezione di eventuali messaggi di richiesta d'acquisto dichiarandosi listener della destinazione.

```
this.queue = << lookup Queue "MokaQueueBuy"  >>
QueueReceiver queueReceiver = queueSession.createReceiver(this.queue);
queueReceiver.setMessageListener(this);
```

L'utilizzo del campo JMSReplyTo dell'header del messaggio è analogo al caso precedente. Il TicketSeller specifica in fase di publishing la coda alla quale inviare la risposta

```
message.setJMSReplyTo(this.queue);
```

mentre il TicketBuyer, una volta ricevuto il messaggio, legge il campo JMSReplyTo e provvede a inviare il messaggio di richiesta d'acquisto

```
Queue buyQueue = (Queue)dest;
// creo il QueueSender
QueueSender queueSender = this.queueSession.createSender(buyQueue);
queueSender.send(msg); // invio il messaggio d'acquisto
```

Una volta implementata la parte relativa all'acquisto dei biglietti, resta da risolvere un ultimo problema: come è possibile identificare l'acquirente tra gli *n* subscriber attivi?

Si potrebbe decidere di specificare l'identità dell'acquirente con un opportuno message property:

```
msg.setStringProperty("ACQUIRENTE",userName);
queueSender.send(msg);
```

o specificarlo direttamente nel corpo del messaggio stesso

Figura 12.11 – *Acquisto dei biglietti: utilizzo del topic temporaneo.*

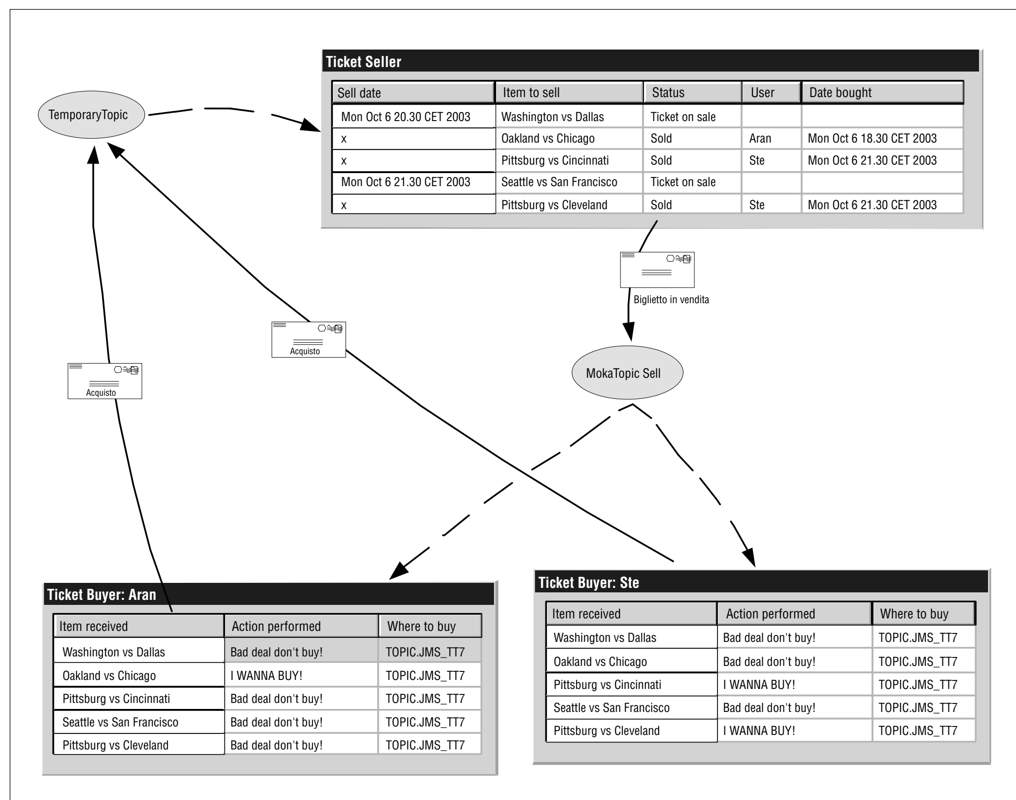
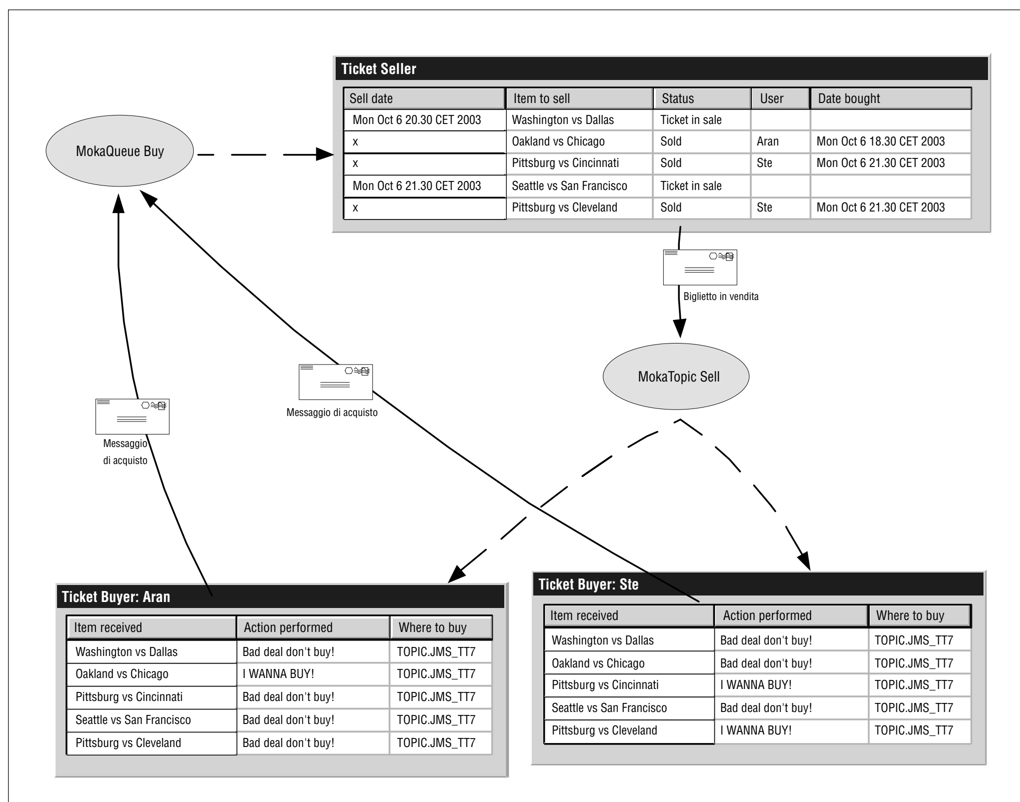


Figura 12.12 – *Acquisto dei biglietti: utilizzo della queue.*




```
ticket.setBuyerId(username);
msg.setObject(ticket);
queueSender.send(msg);
```

Esiste anche la possibilità di utilizzare il campo `JMSCorrelationID` dell'header del messaggio JMS. Tale campo può essere utilizzato come correlatore di messaggi, per esempio per correlare i messaggi di response ai relativi messaggi request

```
public void onMessage(Message message){
    . . .
    message.setJMSCorrelationID(message.getJMSMessageID());
```

```
publisher.publish(message);
```

ma anche con valori di valenza puramente applicativa come nel nostro caso, dove il campo JMSCorrelationID verrà riempito con lo username dell'acquirente prima di inviare l'ordine d'acquisto

```
msg = (ObjectMessage) message;
// Imposto il campo correlationID con lo username del client
msg.setJMSCorrelationID(this.userName);
// invio il messaggio d'acquisto
queueSender.send(msg);
```

Riferimenti bibliografici

JMS Specification
http://java.sun.com/products/jms/docs.html

JMS Tutorial
http://java.sun.com/products/jms/tutorial/index.html

D.A. CHAPPEL – R.M. HAEFEL, *Java Message Service*, O'Reilly

P. GOMEZ – P. ZADROZNY, *Java 2 Enterprise Edition with BEA Weblogic Server*, Wrox

Capitolo 13

JavaMail

MASSIMILIANO BIGATTI

All'interno delle funzionalità offerte dalla piattaforma Java si trova anche il supporto alla posta elettronica, realizzato tramite le API JavaMail. Questa libreria offre le funzionalità necessarie a implementare applicazioni di e-mail completamente funzionanti, come Outlook o Netscape Mail. Non essendo realizzata con componenti visuali – la creazione di una interfaccia utente è interamente a carico dello sviluppatore – è utilizzabile sia in applicazioni desktop sia sul lato server, ad esempio per implementare una WebMail.

Come noto, esistono diversi protocolli che consentono a una applicazione di gestire la posta elettronica. I principali sono tre:

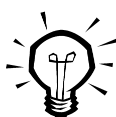
- SMTP (Simple Mail Transfer Protocol): consente l'invio di un messaggio di posta.
- POP (Post Office Protocol): protocollo comunemente utilizzato per la ricezione di messaggi di posta da un server.
- IMAP (Internet Message Access Protocol): protocollo di posta orientato al server e di tipo *drop-and-store*, più evoluto rispetto a POP e dotato di funzionalità di gestione del server che facilitano la gestione della posta off-line.

Attualmente le versioni più diffuse sono la 3 per POP e la 4 per IMAP.

JavaMail, come vedremo, supporta tutti questi protocolli, ma non ne è limitato, in quanto implementa un'architettura a *provider*, che consente di svincolare le applicazioni dal reale protocollo utilizzato, creando un'astrazione di un completo sistema di posta elettronica. In modo similare a quanto avviene in JDBC, dove la comunicazione effettiva al database avviene tramite driver specifici, allo stesso modo JavaMail si astrae dai protocolli specifici, che sono comunque forniti assieme al package principale.

Figura 13.1 – *Tipologie di client di posta elettronica.*





Va notato come le implementazione dei provider di riferimento appartengano al package sun.* e non al package javax.*. Non si tratta quindi di API standard che fanno parte delle specifiche di JavaMail.

Le piattaforme

Il package JavaMail è ottenibile in due diversi modi; per prima cosa è possibile scaricare l'implementazione di riferimento e la documentazione dal sito ufficiale ospitato da Sun, che si trova all'indirizzo http://java.sun.com/products/javamail/. Il package è utilizzabile all'interno di un ambiente J2SE a partire dalla versione 1.1.7, e richiede anche il JavaBeans Activation Framework (http://java.sun.com/products/javabeans/glasgow/jaf.html), un package forse poco famoso, che ha lo scopo di abilitare la gestione, all'interno di applicazioni Java, di blocchi di dati di tipo arbitrario, incapsulandoli e accedendovi, e di scoprire le operazioni che è possibile effettuare su di essi. Esempi di blocchi di dati arbitrari sono le immagini JPEG o GIF, oppure i documenti Word o Excel; l'Activation Framework è utilizzato all'interno di JavaMail per supportare gli allegati binari ai messaggi di posta e anche per scoprire a runtime il tipo di dato memorizzato all'interno di un blocco binario.

Se invece si sta utilizzando la piattaforma Java2 Enterprise Edition (J2EE), a partire dalla versione 1.3, non è necessario scaricare né l'uno, né l'altro package, in quanto i servizi legati alla posta elettronica sono parte fondamentale e formante della piattaforma enterprise.

Ovviamente le modalità d'accesso cambieranno leggermente nelle due piattaforme, in modo similare alle differenze che esistono in JDBC per accedere a una connessione al database in una applicazione J2SE ed in una applicazione J2EE, ma il corpo delle funzionalità rimane il medesimo.

Invio di un messaggio

Per illustrare con semplicità un utilizzo pratico di JavaMail, si osservi il listato riportato più avanti, che implementa un semplice programma di invio di un messaggio di posta, forse il più semplice realizzabile con queste API. La prima cosa che si nota è la presenza dell'importazione dei package javax.mail e javax.mail.internet, due componenti principali di JavaMail.

All'interno del primo package si trovano le classi principali, come Session e Message, utilizzate nel programma per l'invio del messaggio; la prima classe – di tipo final – implementa una sessione di posta e ha lo scopo di raccogliere proprietà e configurazioni e di fornire le sessioni alle classi client. Le sessioni possono essere di due tipi: condivise o meno; nel primo caso un'unica sessione viene utilizzata da parte di più sezioni del programma, tipicamente in una applicazione desktop. In ambiente server è invece preferibile utilizzare sessioni non condivise, ottenute in modo similare a come in JDBC si ottiene una connessione da una DataSource.

Per ottenere una sessione condivisa è necessario utilizzare il metodo Session.getDefaultInstance() a cui è indispensabile passare un oggetto Properties con i parametri di configurazione necessari a operare con il protocollo di posta. L'unico parametro indispensabile in questo caso è mail.smtp.host, che indica il nome o l'indirizzo del server SMTP di invio. Il programma poi crea un oggetto Message di tipo MimeMessage e ne imposta le proprietà. In particolare:

- il mittente;
- il destinatario;
- l'oggetto;
- la data di invio;
- il testo del messaggio.

Questi parametri sono impostati tramite i metodi riassunti in tab. 13.1 che nella classe Message sono astratti, mentre in MimeMessage sono effettivamente implementati.

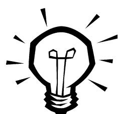
A questo punto è possibile inviare il messaggio utilizzando il metodo Transport.send(). Questo metodo si occuperà dunque di individuare tutti i destinatari (tramite Message.getAllRecipients()) e di inviare loro il messaggio, utilizzando il trasporto appropriato per ciascun destinatario (alcuni di questi, infatti, potrebbero non essere destinatari Internet).

Tabella 13.1 – *Alcuni metodi della classe Message.*

Metodo	Scopo
setFrom()	Imposta il mittente sotto forma di oggetto InternetAddress
setRecipients()	Aggiunge un destinatario all'elenco dei destinatari. Il tipo può essere TO, BCC o CC definiti come campi statici nella classe Message.RecipientType
setSubject()	Imposta l'oggetto del messaggio
setSentDate()	Imposta la data di invio del messaggio
setText()	Imposta il testo del messaggio
setReplyTo()	Imposta l'indirizzo a cui rispondere
setFlag()	Imposta un flag (utilizzato ad esempio per le priorità)

Gli indirizzi sono infatti stati rappresentati con oggetti InternetAddress (classe presente nel package javax.mail.internet), classe che rappresenta indirizzi nella forma *nome@server.dominio* e che estende javax.mail.Address.

Il metodo send() può sollevare due eccezioni nel caso l'invio non vada a buon fine: SendFaildException e MessagingException, entrambi presenti nel package javax.mail.



Si noti che send() può eseguire un invio parziale. Se ad esempio un messaggio è indirizzato a più destinatari – chiamiamoli Mario, Giovanni, Andrea, Luca, Elisa e Maurizio nell'ordine – e l'invio fallisce per Giovanni, Luca e Maurizio, l'invio a Mario, Andrea ed Elisa (chi mancava dall'elenco precedente) viene comunque fatto. Al termine delle operazioni viene sollevata una eccezione che riassume quanto successo: fornisce l'elenco dei destinatari per cui l'invio è andato a buon fine e l'elenco dei destinatari per cui l'invio non è riuscito.

Ed ecco il listato del programma:

```
package com.mokabyte.mokabook2.javamail;

import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;

public class Invio {

  public static void main( String[] args ) {
```

```
        try {
            Properties props = System.getProperties();
            props.put( "mail.smtp.host", args[2] );

            Session session = Session.getDefaultInstance( props );
            Message message = new MimeMessage( session );

            InternetAddress from = new InternetAddress( args[0] );
            InternetAddress to[] = InternetAddress.parse( args[1] );

            message.setFrom( from );
            message.setRecipients( Message.RecipientType.TO, to );
            message.setSubject( args[3] );
            message.setSentDate( new Date() );
            message.setText( args[4] );

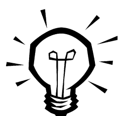
            Transport.send(message);
        } catch(MessagingException e) {
            e.printStackTrace();
        }
    }
}
```

Per eseguire l'esempio è possibile utilizzare il file di Ant build.xml fornito con il codice sorgente, ma è necessario modificare i parametri che riguardano il destinatario e il server SMTP da utilizzare. Per inviare il messaggio di prova digitare poi:

```
ant invio
```

La porzione di script di Ant che si occupa di eseguire il programma è la seguente:

```
<target name="testInvio">
    <java classname="com.mokabyte.mokabook2.javamail.Invio" fork="yes">
        <classpath refid="project-classpath"/>
        <arg value="sender@null.it" />
        <arg value="max@bigatti.it" />
        <arg value="mail.tin.it" />
        <arg value="[Mokabyte] - Messaggio di prova" />
    <arg value="Questo e' un messaggio di prova per verificare l'invio di un messaggio attraverso le API di SUN
JavaMail. Questo package implementa una astrazione di un sistema di posta elettronica
fornendo anche provider per i protocolli SMTP, IMAP e POP3" />
        </java>
</target>
```



Attenzione: per il corretto funzionamento degli esempi presentati, sono necessarie alcune librerie. Eccone un elenco: activation.jar, imap.jar, mailapi.jar, pop3.jar, smtp.jar. Assicurarsi che esse siano state correttamente installate prima di eseguire gli esempi.

Se tutto funziona correttamente si dovrebbe essere in grado di ricevere il messaggio e ottenere qualcosa di simile a quanto presente in fig. 13.2.

Ricevere messaggi

Implementando un insieme di funzionalità completo per la gestione della posta elettronica, JavaMail dispone anche del supporto alla ricezione di messaggi di posta elettronica, tramite l'accesso a server di posta come POP e IMAP.

Figura 13.2 – *Il messaggio ricevuto dal programma "Invio".*



Il supporto alla ricezione dei messaggi è inserito nel più ampio contesto dei folder, contenitori di messaggi che ricalcano concettualmente le caselle di posta dei programmi di e-mail, come la posta in arrivo, la posta in uscita e la posta inviata.

La classe Folder, presente nel package javax.mail, rappresenta un folder astratto che può contenere messaggi (oggetti Message), altri folder o entrambe le categorie di oggetti in modo gerarchico sotto forma di albero. Le sottoclassi di Folder, come IMAPFolder e POP3Folder, completano l'infrastruttura implementando gli specifici protocolli e limitando eventualmente la flessibilità nella costruzione di gerarchie di Folder, nel caso il protocollo specifico non supporti questa possibilità. I Folder lavorano a stretto contatto con gli oggetti Store, che si occupano invece di fornire un modello per la memorizzazione dei messaggi, per il protocollo di accesso e per l'ottenimento dei messaggi. Anche in questo caso Store è una classe astratta usata come superclasse da IMAPStore e POP3Store, le due implementazioni fornite da JavaMail.

Il nome completo di un folder può avere significati diversi in funzione dell'implementazione concreta del folder, anche se il nome INBOX è riservato per indicare il folder principale del server. Si noti comunque che le implementazioni di Store devono obbligatoriamente fornire un INBOX. Se il folder primario per un certo protocollo esiste, si chiamerà INBOX, in caso contrario non ci sarà nessun folder con questo nome e la chiamata Folder.getFolder("INBOX") ritornerà null; per verificare l'esistenza di un folder si consiglia però di utilizzare il metodo Folder.exists().

I passi per leggere da una casella di posta prevedono dunque, dopo l'ottenimento di un oggetto Session, il recupero dell'oggetto Store a partire dalla sessione, tramite il metodo Session.getStore(). Questo ritornerà un oggetto Store in funzione del protocollo configurato nella proprietà mail.store.protocol; in alternativa è possibile chiamare il metodo Session.getStore(string) e fornire direttamente il protocollo desiderato, ad esempio pop3. Se il protocollo richiesto non viene trovato, viene sollevata una eccezione NoSuchProviderException.

Una volta ottenuto l'oggetto Folder che rappresenta la casella richiesta, è necessario aprirla tramite il metodo open(), che si aspetta un parametro modo, che può valere Folder.READ_ONLY e Folder.READ_WRITE; una volta aperto il folder, è possibile leggerne i messaggi con il metodo getMessages(). Una panoramica dei metodi di Folder è riportata in tab. 13.2.

Nel listato riportato poco sotto è presente un esempio di lettura di messaggi che utilizza un server POP3. Come si vede, l'implementazione ricalca la procedura sopra esposta; nel visualizzare i messaggi è stato implementato un controllo sul mittente. Di norma il metodo getPersonal() su un oggetto InternetAddress fornisce il nome del mittente, ma non sempre. Alcune volte il metodo restituisce null ed è dunque necessario rivolgersi al metodo toString() per ottenere una rappresentazione significativa dell'indirizzo.

```
package com.mokabyte.mokabook2.javamail;

import java.util.*;

import javax.mail.*;
import javax.mail.internet.*;

public class Ricezione {
```

Tabella 13.2 – *Metodi della classe Folder.*

Metodo	Scopo
create()	Crea ricorsivamente questo folder all'interno dello Store; se sottofolder di questo folder non esistono, vengono creati anch'essi.
delete()	Cancella, eventualmente in modo ricorsivo, il folder corrente.
exists()	Indica se questo folder esiste nello store.
expunge()	Cancella definitivamente i messaggi marcati come cancellati.
getFullName()	Restituisce il nome completo del folder
getMessage()	Restituisce il messaggio con l'indice indicato.
getMessageCount()	Restituisce il numero di messaggi in questo folder.
getMessages()	Restituisce un array dei messaggi contenuti nel folder, eventualmente restringendone l'insieme, specificando un indice di inizio o fine, oppure un array di indici.
getNewMessageCount()	Restituisce il numero di messaggi nuovi.
getParent()	Restituisce il folder padre di quello attuale.
getSeparator()	Restituisce il carattere utilizzato da questo folder per separare, all'interno del nome completo del folder, i diversi sottofolder.
getUnreadMessageCount()	Restituisce il numero di messaggi non letti nel folder.
getURLName()	Restituisce la rappresentazione completa URL di questo folder senza includere la password di accesso.
open()	Apre il folder.
renameTo()	Cambia il nome di questo folder.
search()	Ricerca, utilizzando un oggetto SearchTerm, i messaggi all'interno del folder.

```
public static void main(String args[]) {

  try {
    Properties props = System.getProperties();
    Session session = Session.getDefaultInstance(props, null);

    Store store = session.getStore("pop3");
    store.connect(args[0], args[1], args[2]);

    Folder folder = store.getDefaultFolder();
    if (folder != null) {

      folder = folder.getFolder("INBOX");
      if (folder != null) {
        folder.open(Folder.READ_ONLY);

        Message[] elencoMessaggi = folder.getMessages();
        for (int indice = 0; indice < elencoMessaggi.length; indice++) {
          Message messaggio = elencoMessaggi[ indice ];
```

```
                InternetAddress fromAddress = (InternetAddress)messaggio.getFrom()[0];
                String from = fromAddress.getPersonal();
                if( from == null ) {
                  from = fromAddress.toString();
                }

                System.out.println(
                  "DA:" + from +
                  " OGGETTO: " + messaggio.getSubject() +
                  " DATA: " + messaggio.getSentDate()
                );
              }

              folder.close(false);
            } else {
              System.out.println( "Folder non trovato" );
            }
          } else {
            System.out.println( "Folder di default non trovato" );
          }
          store.close();
        } catch (Exception ex) {
          ex.printStackTrace();
        }
      }

    }
```

Per eseguire l'esempio, utilizzare il comando:

```
Ant ricezione
```

La porzione di script di Ant eseguita è la seguente (attenzione anche in questo caso ad indicare i parametri di configurazione corretti):

```
<target name="testRicezione">
  <java classname="com.mokabyte.mokabook2.javamail.Ricezione" fork="yes">
    <classpath refid="project-classpath"/>
    <arg value="pop3.server.it" />
    <arg value="max/bigatti.it" />
    <arg value="password" />
  </java>
</target>
```

Un tipico output è simile al seguente (riformattato):

```
DA:Apple Developer Connection
OGGETTO: ADC News #372
DATA: Sun Oct 26 03:37:20 CET 2003

DA:Apple iCards
OGGETTO: Massimiliano Bigatti has sent you an Apple iCard
DATA: Thu Oct 30 10:17:28 CET 2003

DA:mrossi@tiscalinet.it
OGGETTO: orologio a mano...
DATA: Thu Oct 30 18:03:12 CET 2003

DA:sender@null.it
OGGETTO: [Mokabyte] - Messaggio di prova
DATA: Thu Oct 30 18:49:52 CET 2003
DA:sender@null.it
OGGETTO: [Mokabyte] - Messaggio di prova
DATA: Thu Oct 30 18:50:48 CET 2003
```

Eventi sui folder

Le API JavaMail implementano anche alcuni eventi; le operazioni sui folder possono essere infatti controllate da altri oggetti tramite una serie di eventi. Ad esempio, se una parte dell'applicazione rimuove un messaggio, i listener registrati sul folder stesso (e sullo Store da cui il folder deriva), ne ricevono notifica. In questo modo è possibile eseguire porzioni di codice arbitrario a fronte di determinati eventi, quali ad esempio la cancellazione dei messaggi.

Di seguito sono riportati i metodi per l'aggiunta dei listener nelle classi Folder:

```
public void addConnectionListener (ConnectionListener  l)
public addFolderListener (FolderListener  l)
public void addMessageChangedListener (MessageChangedListener  l)
public void addMessageCountListener (MessageCountListener  l)
```

mentre i metodi necessari all'aggiunta dei listener della classe Store sono i seguenti:

```
public addFolderListener (FolderListener  l)
public void addStoreListener (StoreListener  l)
```

Quando avviene una modifica al contenuto del folder, viene generato un evento FolderEvent appropriato. Ad esempio, quando viene invocato il metodo delete() su un Folder, viene generato un evento con type = DELETED. La classe FolderEvent dispone infatti dei metodi:

- getFolder(): restituisce il folder coinvolto nell'operazione;
- getNewFolder(): se l'operazione eseguita è stata una rinominazione, questo metodo restituisce un folder che rappresenta il nuovo nome;
- getType(): restituisce il tipo di operazione, che può essere FolderEvent.CREATED, FolderEvent.DELETED, FolderEvent.RENAMED;

Nel listato che segue è presente un esempio di utilizzo degli eventi; in questo caso viene utilizzato l'evento MessageCountEvent, per individuare l'arrivo di un nuovo messaggio e realizzare così un programma monitor che segnali all'utente che ha nuova posta.

Come si nota osservando il codice, viene fatto uso del metodo addMessageCountListener() presente nella classe Folder, a cui viene passato un nuovo oggetto anonimo derivato da MessageCountAdapter. Questo implementa il metodo messagesAdded(), metodo che indica l'aggiunta di messaggi al Folder; quando questo metodo viene richiamato, viene stampata a console la lunghezza dell'array di messaggi contenuti nell'oggetto evento, di classe MessageCountEvent. Questo array identifica appunto i messaggi coinvolti dall'operazione, che può essere di aggiunta o rimozione, come indicato dal metodo getType(), che può restituire ADDED o REMOVED.

```
package com.mokabyte.mokabook2.javamail;

import java.util.*;

import javax.mail.*;
import javax.mail.event.*;

public class Monitor {

  static final int millisecAttesa = 4500;

  public static void main(String args[]) {

    try {
      Properties props = System.getProperties();
      Session session = Session.getDefaultInstance(props, null);

      Store store = session.getStore("imap");
      store.connect(args[0], args[1], args[2]);

      Folder folder = store.getDefaultFolder();
      if (folder != null) {

        folder = folder.getFolder("INBOX");
```

```
            if (folder != null) {

              folder.open(Folder.READ_WRITE);

              folder.addMessageCountListener(new MessageCountAdapter() {
                public void messagesAdded(MessageCountEvent ev) {
                  Message[] msgs = ev.getMessages();
                  System.out.println("Sono arrivati " + msgs.length + " nuovi messaggi");
                }
              });

              System.out.println( "Pronto a ricevere messaggi" );
              for(;;) {
                Thread.sleep( millisecAttesa );

                folder.getMessageCount();
                System.out.println( "ping..." );
              }
            } else {
              System.out.println( "Folder non trovato" );
            }
          } else {
            System.out.println( "Folder di default non trovato" );
          }
        } catch (Exception ex) {
          ex.printStackTrace();
        }
      }

    }
```

Per eseguire il codice si deve digitare:

```
ant monitor
```

La porzione di script di Ant che esegue il comando è la seguente:

```
<target name="testMonitor">
  <java classname="com.mokabyte.mokabook2.javamail.Monitor" fork="yes">
    <classpath refid="project-classpath"/>
    <arg value="pop3.server.it" />
    <arg value="max/bigatti.it" />
    <arg value="password" />
```

```
  </java>
</target>
```

Messaggi multipart

In precedenza è stato utilizzato il metodo setText() per impostare il contenuto di un messaggio, ma in realtà questo è un metodo "di convenienza", che effettua alcune operazioni per impostare il tipo di contenuto (MIME) a text/plain. JavaMail supporta infatti messaggi di tipo multipart, composti cioè da contenuti multipli di tipi eterogenei. Un messaggio di posta potrebbe essere composto infatti da un contenuto in testo semplice e da un contenuto HTML; spesso questi messaggi di tipo ricco possono essere visualizzati in un modo o nell'altro in funzione delle capacità dei singoli programmi di client email. Un altro uso dei messaggi multipart è quello legato agli allegati; quando a un messaggio di posta si allega un file di Word o Excel, un file compresso o un'immagine, il client crea un messaggio multipart con una parte per ciascun allegato, associando alla parte il tipo MIME dell'archivio (per esempio image/gif) e il contenuto binario del file. È in questo momento che entra in gioco il JavaBeans Activation Framework, il quale offre appunto il supporto richiesto a queste operazioni con dati binari, permettendo addirittura l'invio di oggetti Java serializzati!

Invio di allegati

Per inviare un messaggio multipart in JavaMail le operazioni di connessione al provider e di inizializzazione del messaggio sono le medesime di quelle viste nel primo listato presentato; le novità intervengono solo nella composizione del corpo da inviare. È necessario infatti utilizzare un oggetto Multipart, che consente di memorizzare diverse parti di corpo (oggetti BodyPart) e che fornisce metodi per impostarle e restituirle; anche in questo caso Multipart è una classe astratta che demanda alle sottoclassi l'effettiva realizzazione di parte dei suoi servizi; per la posta Internet è disponibile la classe MimeMultipart che si basa appunto sulle convenzioni MIME.

La classe MimeMultipart definisce diversi sottotipi, conformi alle specifiche MIME, che possono essere "mixed", "alternative", "related" e così via; il tipo principale è invece "multipart". Questo approccio consente una più agevole corrispondenza nell'Activation Framework, disaccoppiando il processo di fornitura dei gestori delle diverse parti dalle API JavaMail.

Nel listato riportato più sotto è presente il codice completo di un programma di prova che esegue un invio di un messaggio con allegato, sviluppato a partire dalla classe Invio presentata nel primo listato illustrato nel capitolo. Come si nota osservando il codice, viene per prima cosa istanziato un oggetto MimeMultipart, che ospiterà il corpo del messaggio e subito dopo un oggetto MimeBodyPart (sottoclasse di BodyPart) che fornisce una implementazione MIME di una singola parte. Nell'esempio vengono utilizzate due parti, una per il testo del messaggio e un'altra per l'allegato Word. Se nel primo caso è sufficiente chiamare il metodo setText() per impostare il contenuto della parte, che è solo testuale, nel secondo caso è necessario utilizzare le classi di Activation Framework per ottenere un blocco dati binario che contenga il documento Word.

In particolare, viene creato un nuovo DataSource a partire dal file Documento.doc che viene letto tramite un'istanza di DataHandler:

```
DataSource source = new FileDataSource( "Documento.doc" );
messageBodyPart = new MimeBodyPart();
messageBodyPart.setDataHandler( new DataHandler(source) );
messageBodyPart.setFileName( "Documento.doc" );
```

I due oggetti MimeBodyPart vengono poi aggiunti all'oggetto MimeMultipart tramite il metodo addBodyPart().

Per eseguire l'esempio, digitare:

```
ant invioMultipart
```

La porzione di script Ant invocata è la seguente:

```
<target name="testInvioMultipart">
  <java classname="com.mokabyte.mokabook2.javamail.InvioMultipart" fork="yes">
    <classpath refid="project-classpath"/>
    <arg value="mail.tin.it" />
  </java>
</target>
```

Si dovrebbe ottenere un risultato simile a quello presente in fig. 13.3.

Ed ecco il listato completo relativo a InvioMultipart.java:

```
package com.mokabyte.mokabook2.javamail;

import java.util.*;

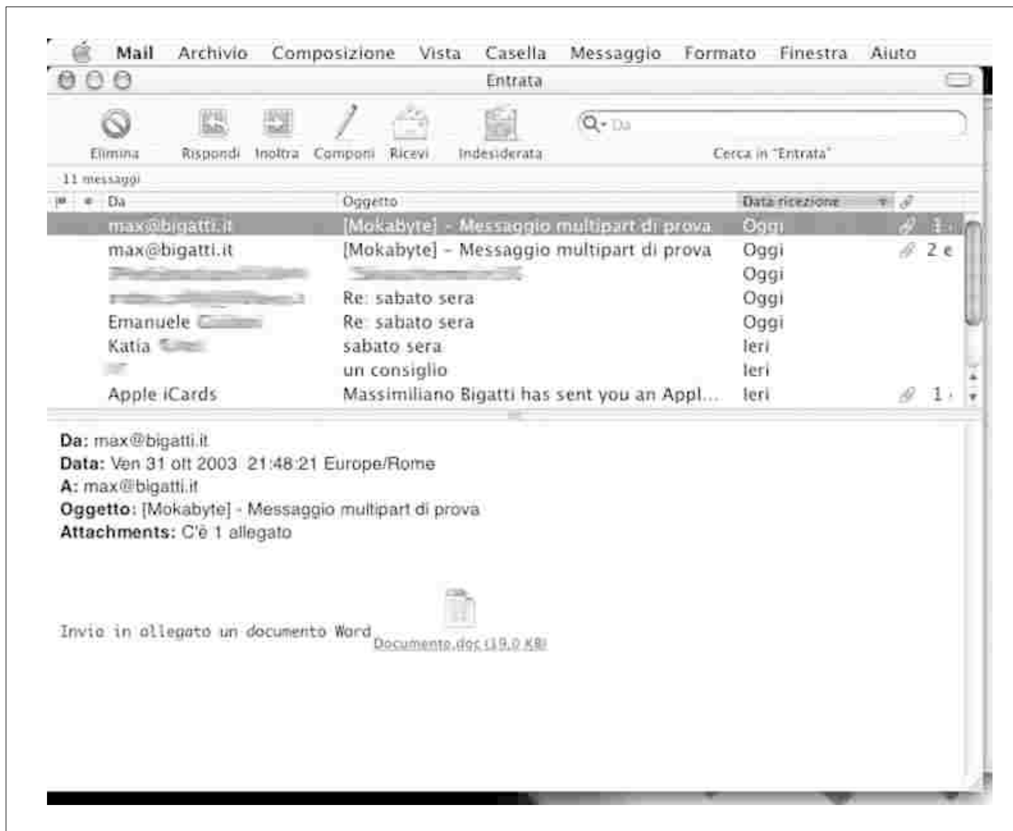
import javax.activation.*;
import javax.mail.*;
import javax.mail.internet.*;

public class InvioMultipart {

  public static void main( String[] args ) {
    try {
      Properties props = System.getProperties();
      props.put( "mail.smtp.host", args[0] );

      Session session = Session.getDefaultInstance( props );
      Message message = new MimeMessage( session );
```

Figura 13.3 – *Il messaggio multipart è stato ricevuto correttamente.*




```
InternetAddress from = new InternetAddress( "max@bigatti.it" );
InternetAddress to[] = InternetAddress.parse( "max@bigatti.it" );

message.setFrom( from );
message.setRecipients( Message.RecipientType.TO, to );
message.setSubject( "[Mokabyte] - Messaggio multipart di prova" );
message.setSentDate( new Date() );

Multipart multipart = new MimeMultipart();

//crea la parte testuale
BodyPart messageBodyPart1 = new MimeBodyPart();
messageBodyPart1.setText("Invio in allegato un documento Word");
```

```
            //crea l'allegato Word
            DataSource source = new FileDataSource( "Documento.doc" );
            BodyPart messageBodyPart2 = new MimeBodyPart();
            messageBodyPart2.setDataHandler( new DataHandler(source) );
            messageBodyPart2.setFileName( "Documento.doc" );

            //aggiunge le parti all'oggetto multipart
            multipart.addBodyPart( messageBodyPart1 );
            multipart.addBodyPart( messageBodyPart2 );

            //imposta come contenuto del messaggio l'oggetto multipart
            message.setContent(multipart);

            Transport.send(message);
        } catch(MessagingException e) {
            e.printStackTrace();
        }
    }
}
```

Ricezione di allegati

La ricezione di messaggi con allegati passa, in modo similare all'invio, dalle classi Multipart e Part. Nell'esempio riportato nel listato seguente, una volta ottenuto il messaggio, esso viene stampato tramite il metodo stampaMessaggio() che limita la sua elaborazione ai soli messaggi multipart. Tramite il metodo getCount() sull'oggetto Multipart ottenuto con la chiamata Message.getContent() si ottiene il numero di parti di cui è composto l'oggetto; queste vengono estratte una a una tramite il metodo getBodyPart(). A questo punto viene chiamato il metodo stampaParte() che elabora ciascuna parte; per prima cosa viene ottenuto il tipo MIME, tramite il metodo getContentType(). Se questo è di tipo text/plain, il contenuto è semplice testo, che può essere stampato direttamente a console riga per riga tramite la classe BufferedReader; in caso di allegati binari, viene creato un file su disco tramite la classe FileOutputStream.

```
package com.mokabyte.mokabook2.javamail;

import java.io.*;
import java.util.*;

import javax.mail.*;
import javax.mail.internet.*;

public class RicezioneMultipart {
```

```
public static void main(String args[]) {

   try {
      Properties props = System.getProperties();
      Session session = Session.getDefaultInstance(props, null);

      Store store = session.getStore("pop3");
      store.connect(args[0], args[1], args[2]);

      Folder folder = store.getDefaultFolder();
      if (folder != null) {

         folder = folder.getFolder("INBOX");
         if (folder != null) {
            folder.open(Folder.READ_ONLY);

            Message[] elencoMessaggi = folder.getMessages();
            for (int indice = 0; indice < elencoMessaggi.length; indice++) {
               Message messaggio = elencoMessaggi[ indice ];

               InternetAddress fromAddress = (InternetAddress)messaggio.getFrom()[0];
               String from = fromAddress.getPersonal();
               if( from == null ) {
                  from = fromAddress.toString();
               }

               System.out.println( "----------------------------------------" );
               System.out.println(
                  "DA:" + from +
                  " OGGETTO: " + messaggio.getSubject() +
                  " DATA: " + messaggio.getSentDate() +
                  "\n"
               );

               stampaMessaggio( messaggio );
            }

            folder.close(false);
         } else {
            System.out.println( "Folder non trovato" );
         }
      } else {
         System.out.println( "Folder di default non trovato" );
```

```
      }
      store.close();
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  static void stampaMessaggio( Message msg ) throws MessagingException, IOException {
    Part msgPart = msg;
    Object contenuto = msgPart.getContent();
    if( contenuto instanceof Multipart ) {
      Multipart mp = (Multipart)contenuto;

      for( int i=0; i<mp.getCount(); i++ ) {
        stampaParte( mp.getBodyPart(i), i );
      }
    }
  }

  static void stampaParte( Part parte, int count ) throws MessagingException, IOException {
    String contentType = parte.getContentType();

    System.out.println( "Disposizione: " + parte.getDisposition() );
    System.out.println( "Testo: " );

    if( contentType.startsWith("text/plain") ) {

      InputStream in = parte.getInputStream();
      BufferedReader reader = new BufferedReader(
        new InputStreamReader( in )
      );

      do {
        String linea = reader.readLine();
        if( linea == null ) {
          break;
        }

        System.out.println( linea );
      } while( true );

      reader.close();
      in.close();
```

```
        } else {

            String filename = parte.getFileName();
            if( filename == null ) {
                filename = "allegato" + count + ".bin";
            }
            FileOutputStream writer = new FileOutputStream( filename );

            byte[] buffer = new byte[ 4096 ];
            InputStream in = parte.getInputStream();

            while( true ) {
                int readed = in.read( buffer );
                if( readed == -1 ) {
                    break;
                }

                writer.write( buffer, 0, readed );
            }

            writer.close();
            in.close();

            System.out.println("Salvato il file " + filename );
        }
    }

}
```

Per provare questo esempio digitare:

```
ant ricezioneMultipart
```

La porzione di script di Ant relativo è la seguente:

```
<target name="testRicezione">
    <java classname="com.mokabyte.mokabook2.javamail.RicezioneMultipart" fork="yes">
        <classpath refid="project-classpath"/>
        <arg value="pop3.server.it" />
        <arg value="max/bigatti.it" />
        <arg value="password" />
    </java>
</target>
```

Un esempio di output è il seguente (riformattato):

```
----------------------------------------
DA:Apple iCards
OGGETTO: Massimiliano Bigatti has sent you an Apple iCard DATA: Thu Oct 30 10:17:28 CET 2003

Disposizione: null
Testo:
Salvato il file allegato0.bin
Disposizione: inline
Testo:
Salvato il file iCard.jpg
Disposizione: null
Testo:
Salvato il file allegato2.bin
----------------------------------------
DA:max@bigatti.it
OGGETTO: [Mokabyte] - Messaggio multipart di prova DATA: Fri Oct 31 21:50:02 CET 2003

Disposizione: null
Testo:
Invio in allegato un documento Word
Disposizione: attachment
Testo:
Salvato il file Documento.doc
```

Conclusioni

Le API JavaMail supportano un insieme completo di funzionalità per la gestione della posta elettronica; oltre all'invio e ricezione di messaggi semplici e multipart, sono presenti funzionalità di gestione della quota dell'utente, per controllare l'occupazione di spazio delle caselle di posta, per la ricerca di messaggi all'interno delle caselle postali e per l'autenticazione dell'utente.

Riferimenti bibliografici

[1]
JavaMail QuickStart
http://www.javaworld.com/javaworld/jw-10-2001/jw-1026-javamail.html

[2]
Managing ezines with JavaMail and XSLT, Part 2
http://www-106.ibm.com/developerworks/xml/library/x-xmlist2/?open&l=842%2Ct=gr%2Cp=JavaMail2

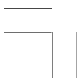
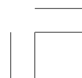
[3]
Fundamentals of the JavaMail API short course
http://developer.java.sun.com/developer/onlineTraining/JavaMail/contents.html

[4]
JavaMail Homepage
http://java.sun.com/products/javamail/

[5]
JavaMail API documentation
http://java.sun.com/products/javamail/javadocs/index.html

[6]
JavaBeans Activation Framework
http://java.sun.com/products/javabeans/glasgow/jaf.html



Appendice A

Applet

STEFANO ROSSINI – LUCA DOZIO

Che cosa è un'applet?

Un'applet è un particolare programma Java inserito in una pagina HTML scaricato dinamicamente dalla rete ed eseguito dalla Java virtual machine del browser.

Le applet sono state di fondamentale importanza nella storia e nella diffusione di Java rappresentando il punto di forza di Sun per fare breccia nel mercato. Sfruttando l'esistenza di browser per le differenti piattaforme hardware e software Sun ha utilizzato le applet per dimostrare, con successo, la portabilità Java. Le applet hanno da questo punto di vista dato nuova vitalità ai web browser ed in generale alle applicazioni web, rivalutandone le potenzialità.

Grazie al meccanismo del download automatico dalla rete, un'applet può essere eseguita senza la necessità di installare software né sviluppare particolari client per le diverse piattaforme: la rete diviene un canale da cui prelevare il software, ed il browser un client che esegue un programma in locale.

L'inserimento all'interno della pagina HTML di un'applet avviene per mezzo del tag `<APPLET>` con il quale tra l'altro si specifica la classe Java che costituisce l'applet stessa.

Al fine di garantire un elevato margine di sicurezza le applet devono rispettare dei ben precisi vincoli di sicurezza imposti dal Security Manager della macchina virtuale del browser.

Attualmente tutti i browser incorporano una macchina virtuale e sono in grado di eseguire un'applet. Purtroppo si riscontrano sostanziali differenze sia fra le varie marche di browser che fra le varie versioni della stessa casa produttrice (Microsoft e Netscape le maggiori). Queste differenti scelte tecnologiche (e politiche) delle aziende possono causare di fatto un comportamento diverso di un'applet in esecuzione sui vari browser in circolazione.

Lo scopo di questo capitolo è quello di affrontare i principali aspetti legati alla programmazione delle applet.

Il ciclo di vita di un'applet

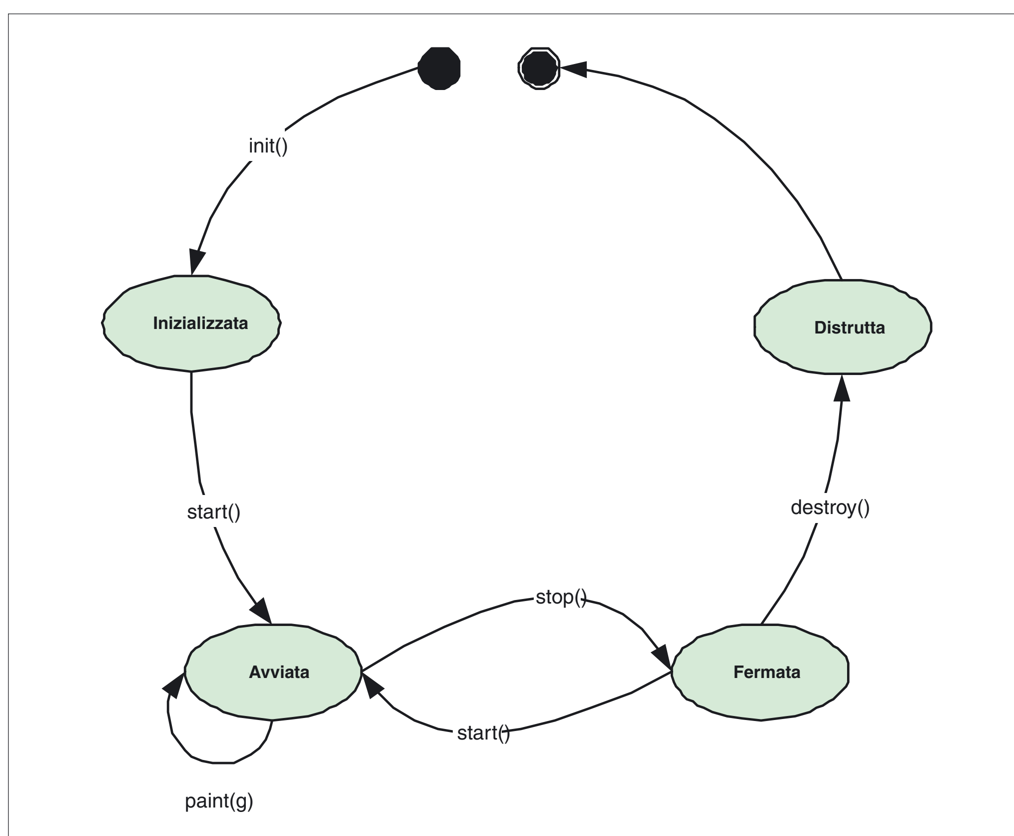
I package Java sono collezioni di classi java correlate tra loro, che il programmatore può utilizzare a piacimento all'interno dei propri programmi.

I package come AWT, Swing ed Applet hanno un'impostazione molto diversa, dal momento che la sequenza delle chiamate ai metodi è predefinita: questo modello di implementazione, in cui il programmatore viene chiamato ad estendere al fine di completare un'applicazione già definita, prende il nome di framework.

Per creare un'applet si deve necessariamente estendere la classe `Applet` (grafica AWT) o `JApplet` (grafica Swing) e ridefinire gli opportuni metodi d'interesse al fine di specializzare il comportamento della propria applet.

Prima di procedere allo sviluppo del framework relativo alle applet è necessario conoscere il loro ciclo di vita, cioè in corrispondenza di quali eventi il framework effettua in modo automatico la chiamata ad opportuni metodi.

Figura A.1 – *Ciclo di vita di un'applet*



I metodi del framework delle applet che vengono invocati automaticamente dalla macchina virtuale del browser sono init(), start(), stop() e destroy().

Il metodo init() viene invocato una volta avvenuto il caricamento e la verifica del bytecode dell'applet al fine di permettere la sua inizializzazione.

Ridefinendo tale metodo è possibile inserire il codice per effettuare le operazioni di inizializzazione come la lettura dei parametri dichiarati dai tag HTML PARAM, caricare le risorse (immagini, suoni, file di testo , ...) e aggiungere componenti grafiche per creare l'interfaccia grafica d'utente (GUI).

Il metodo start() viene invocato dopo che la JVM ha invocato il metodo init() ed ogni qualvolta l'applet appare nel browser; ciò significa che il metodo start() può essere chiamato ripetutamente. Solitamente è il punto in cui viene avviato il thread dell'applet, dove sono creati i thread aggiuntivi (ad esempio per effettuare animazioni o una qualsiasi operazione da eseguire in "parallelo" all'esecuzione dell'applet) e dove sono effettuate tutte le operazioni chiuse dal metodo stop().

Il metodo stop()viene invocato quando l'utente esce dalla pagina in cui si trova l'applet. Può quindi essere chiamato ripetutamente ed ha lo scopo di consentire l'interruzione delle attività avviate nel metodo start() evitando il rallentamento del sistema quando l'utente non sta visualizzando l'applet.

Il metodo destroy() viene chiamato dopo che l'applet è stata fermata mediante il metodo stop nel momento in cui il browser viene chiuso.

Il metodo paint() (definito nella classe Container) permette di "decorare" l'area occupata dall'applet all'interno della pagina HTML.

Il metodo paint() è chiamato quando l'applet ha necessità di aggiornare il suo stato grafico, ad esempio in seguito alla sua visualizzazione sullo schermo per la prima volta, o perché qualche altra finestra si è sovrapposta a quella del browser o perché la stessa è stata ridimensionata.

Nonostante l'applet non deve obbligatoriamente definire il metodo main come in una normale applicazione Java (il suo entry-point è il metodo init), può essere utile scrivere il metodo main per "riusare" l'applet come applicazione stand-alone e/o per scopi di test.

```
public class MyApplet extends Applet {
  boolean isStandalone = false;
  . . .
```

Tabella A.1 – *Descrizione dei metodi invocati nel ciclo di vita di un'applet*

Metodo	Invocazione	Operazione
init()	Al caricamento dell'Applet	Inizializzazione dell'Applet
start()	Dopo il metodo init e ad ogni visualizzazione della pagina HTML	Avvio o riavvio di thread
stop()	Prima di destroy e ogni volta che si cambia pagina	Arresto di thread
destroy()	Alla chiusura del browser	Rilascio delle risorse

```
    public static void main(String[] args){
      MyApplet applet = new MyApplet();
      applet.isStandalone=true;
      Frame frame = new Frame("MyApplet");
      frame.add(applet);
      applet.init();
      frame.setVisible(true);
    }
  . . .
```

Sviluppo di applet

Vediamo un esempio di un' applet che visualizza il messaggio "HelloWorld!". Il codice dell'Applet HelloWorld è il seguente:

```
package com.mokabyte.mokabook.applets;
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorld extends Applet {

 /** Messaggio che l'Applet visualizza */
 private String message;

 /** Le operazioni di inizializzazione dell' applet */
 public void init(){
    this.message = "HelloWorld!";
 }

/**
 * Disegna un rettangolo delle dimensioni specificate(drawRect)
 * Disegna una stringa nella posizione specificata (drawString)
 * @param g oggetto da visualizzare
 */
 public void paint(Graphics g){
   g.drawRect(0, 0, getSize().width-1,getSize().height-1);
   g.drawString(message, 5, 15);
 }
}
```

Come si può notare la visualizzazione del messaggio nel contesto grafico dell' applet nella pagina html, avviene ridefinendo il metodo paint(), ed invocando i metodi drawRect e drawString sull'oggetto di classe Graphics.

Per poter eseguire l'applet è necessario preparare un file HTML in cui sia inserito un riferimento tramite il tag CODE al file .class ottenuto al termine del processo di compilazione.

Di seguito è riportata una porzione di tale file in cui si effettua l'invocazione all'applet nella sua forma più essenziale, garantendo l'aggiunta e la descrizione di ulteriori tag utilizzabili, quando se ne presenterà la necessità:

```
<APPLET CODE=com.mokabook.applets.HelloWorld.class
WIDTH=400
HEIGHT=300>
</APPLET>
```

I tag WIDTH e HEIGHT determinano le dimensioni dello spazio grafico dell'Applet, spazio che non è modificabile a runtime.

Lanciando da browser o mediante l'AppletViewer (tool che emula un browser disponibile nel JDK), il caricamento della pagina HTML, dovrebbe venire visualizzata l'applet con il messaggio "HelloWorld!", in una finestra delle dimensioni dichiarate. Se così non è o il browser è datato e non supporta Java, oppure il browser ha la virtual machine disabilitata.

I tag CODE, WIDTH e HEIGHT sono gli unici indispensabili per il caricamento dell'applet, e non possono essere tralasciati. Altri tag HTML opzionali sono ALIGN (indica l'allineamento dell'applet nella pagina HTML), ARCHIVE (indica il file di archivio .jar nel quale cercare le classi dell'Applet), CODEBASE (indica il percorso da webroot dove ricercare il file .class dell'applet), NAME (assegna all'applet un nome referenziabile da JavaScript o da AppletContext), PARAM definisce i parametri da passare all'applet.

Mediante il tag PARAM è possibile passare dei parametri da pagina HTML all'applet.
Infatti, racchiusi fra i tag <APPLET> e </APPLET>, si possono definire nomi (tag NAME) e valori (tag VALUE) di parametri che l'applet può leggere. Ad esempio

```
<APPLET CODE=com.mokabyte.mokabook.applets.AppletWithParameter.class      CODEBASE=classes
WIDTH=400
HEIGHT=300>
  <PARAM NAME=message VALUE="Ciao a tutti!">
</APPLET>
```

Da dentro l'applet si può accedere ai parametri passati da file HTML utilizzando il metodo getParameter specificando come parametro il nome dell'attributo HTML da leggere:

```
this.message = getParameter("message");
```

Per fornire all'Applet una semplice interfaccia grafica, si può creare un oggetto java.awt.Label contenente il messaggio da visualizzare.

```
import java.applet.Applet;
```

```
import java.awt.Label;
public class MyAwtApplet extends Applet {
. . .
 public void init(){
  this.message = getParameter("message");
  Label lbl = new Label(this.message);
  this.add("Center",lbl);
  . . .
```

Per sfruttare nelle applet le potenzialità delle classi Swing bisogna estendere, non più dalla classe java.applet.Applet, ma dalla classe javax.swing.JApplet. Per quel che riguarda la costruzione dell'interfaccia grafica l'aggiunta di componenti non viene più fatta direttamente sul contenitore dell'Applet (metodo add()), ma sul ContentPane, un pannello di contenimento accessibile attraverso il metodo getContentPane() e delegato a contenere i componenti grafici.

```
import javax.swing.JApplet;
import javax.swing.JLabel;

public class MySwingApplet extends JApplet {
. . .
 public void init(){
  this.message = getParameter("message");
  JLabel lbl = new JLabel(this.message);
  this.getContentPane().add("Center",lbl);

  . . .
```

Per eseguire applet che utilizzino le classi Swing il browser deve supportare nativamente le API Java 2 o avere installato un plugin. Il Java Plug-in è un tool che permette al browser di utilizzare una virtual machine esterna, e quindi di supportare versione del JDK differenti da quella della JVM incapsulata nel browser stesso.

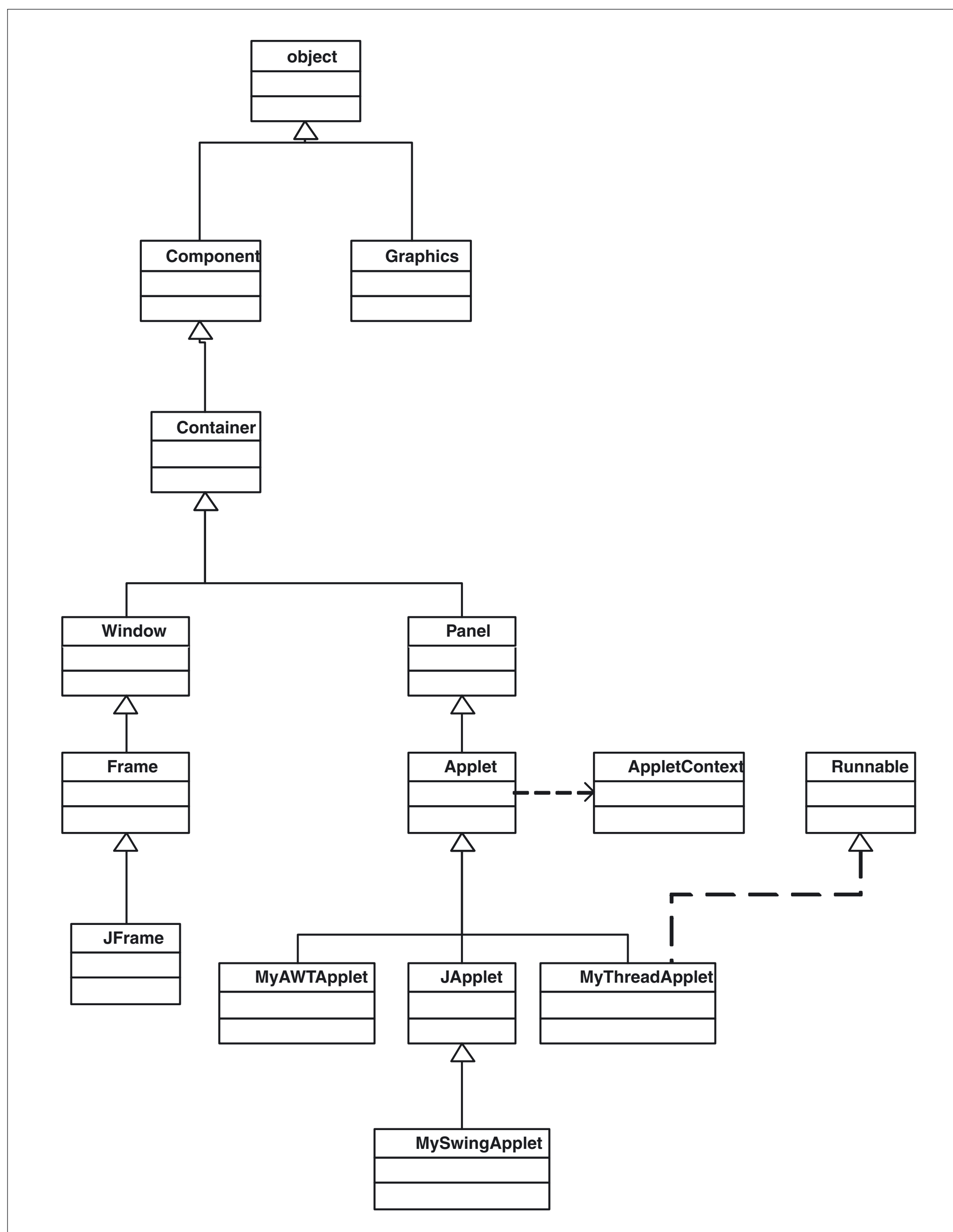
Per specificare l'uso del plugin si rende necessario l'utilizzo di tag specifici al posto del tag <APPLET> nei diversi browser (ad esempio per il browser Microsoft Internet Explorer bisogna utilizzare il tag <OBJECT>).

AppletContext

Un'applet può chiedere al browser di eseguire una serie di operazioni come riprodurre un audioclip, visualizzare un breve messaggio nella riga di stato, visualizzare un'altra pagina web, …

Queste operazioni sono eseguite mediante l'utilizzo dell'interfaccia AppletContext la cui implementazione può essere considerata come un canale di comunicazione tra l'applet e il browser locale.

Figura A.2 – *Diagramma UML della classe Applet e JApplet*



Il metodo getAppletContext() della classe Applet restituisce un reference all'AppletContext tramite il quale è possibile invocare i suoi metodi.
Mediante il metodo void showStatus(String message), è possibile mostrare un messaggio nella riga di stato del browser

```
getAppletContext().showStatus("caricamento immagini");
```

Con il metodo void showDocument(URL url) si visualizza una nuova pagina web

```
URL newUrl = new URL("http://www.mokabyte.it");
getAppletContext().showDocument(newUrl);
```

Con il metodo void showDocument(URL url, String target) è possible indicare al browser la modalità di visualizzazione della nuova pagina ("_self" indica il frame corrente, "_blank" indica una nuova finestra senza nome, …)
Il metodo Image getImage(URL url) permette di ottenenere un'immagine a partire dell'URL specificato così come il metodo AudioClip getAudioClip(URL url) permette di caricare un file audio.
Se una pagina web contiene più di un'applet aventi lo stesso codebase, queste possono comunicare tra loro.
Assegnando al tag NAME di ciascuna applet nel file HTML un nome, è possibile utilizzare il metodo getApplet(String name) dell'interfaccia AppletContext per ottenere un riferimento all'applet

```
Applet appletRef = getAppletContext().getApplet("YourApplet");
```

tramite il quale è possibile invocare i metodi dell'applet una volta effettuato il cast appropriato.

```
((MyApplet)appletRef).metodo();
```

È possibile anche elencare tutte le applet contenute in una pagina HTML dotate o meno di un tag NAME, utilizzando il metodo getApplets() che restituisce un oggetto di classe Enumeration.

```
Enumeration e = getAppletContext().getApplets();
while(e.hasMoreElements()){
  System.out.println("" + e.nextElement());
}
```

Il metodo getApplets() restituisce solo le applet appartenenti allo stesso contesto. Per permettere la comunicazione fra applet appartenenti a contesti differenti si può sfruttare il fatto che la virtual machine del browser è condivisa tra le pagine HTML della medesima sessione. Si può quindi creare una classe che con proprietà e metodi di accesso statici, memorizzi i reference delle applet in esecuzione in pagine differenti al fine di renderli utilizzabili da applet di contesti diversi.

La sicurezza

Il modello di funzionamento di base di un'applet, noto con il nome di network computing porta ad un processo di installazione ed esecuzione di codice remoto e di fatto sconosciuto sulla macchina locale. Il modello cosiddetto sandbox permette l'esecuzione delle applet all'interno di un ambiente protetto in modo da impedirne l'accesso al sistema sottostante. La sandbox è quindi una specie di contenitore dal quale le applet non possono evadere: il controllo sulle operazioni permesse è effettuato dal cosiddetto Security Manager, il quale verifica sia il bytecode in fase di caricamento (*bytecode verifier*), sia in fase d'esecuzione (*security manager*).

Il Security Manager della macchina virtuale del browser ha il compito di lanciare un'eccezione di tipo SecurityException ogni qualvolta un'applet cerca di violare una delle regole di sicurezza imposte dalla sandbox.

Ad esempio le applet non possono accedere al fileystem locale, utilizzare procedure native, leggere la totalità delle proprietà di sistema della macchina sulla quale sono in esecuzione (mentre è possibile leggere proprietà come il file.separator, path.separator, os.name, ma è vietato leggere proprietà come user.name, user.home, …) accettare connessioni da una macchina client, e così via.

Un'applet può quindi prelevare file soltanto dall'host di provenienza. Per non incorrere in errori di sviluppo è importante utilizzare i metodi getDocumentBase() o getCodeBase(): il primo recupera l'URL della pagina contenente l'applet mentre il metodo getCodeBase() recupera l'URL della directory contenente l'applet stessa.

Si riportano alcune esempi di significativi su come usare i due metodi:

```
Image img=getImage(getDocumentBase(),"images/"+"figura.jpg");

AudioClip audioClip=getAudioClip(getDocumentBase(),"audio.mid");

Socket socket = new Socket(getCodeBase().getHost(), 1666);

MyServerInterface serverRef =
(MyServerInterface)Naming.lookup("//" +
  getDocumentBase().getHost()+"/MyService");
```

I vincoli del modello Sandbox, con l'avvento della versione 1.1 e 1.2 del JDK, sono stati "rilassati" permettendo allo sviluppatore delle applet una maggior libertà d'azione introducendo il concetto di applet firmata.

Lo sviluppo di applet firmate può essere utile ad esempio laddove occorre utilizzare dispositivi connessi alla postazione di lavoro locale (scanner, lettori ottici, stampanti, …) o per collegarsi a server diversi da quello da cui è stata scaricata l'Applet stessa.

Lo stesso JDK fornisce i tool per permettere la firma della applet (keytool, jarsigner, …).

Per una trattazione completa dell'argomento, si rimanda al Tutorial Sun http://java.sun.com/docs/books/tutorial.

